Linked List

An ***array*** is a very useful data structure provided in programming languages. However, it has at least two limitations:

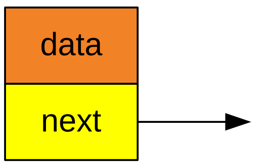
- its size has to be known at compilation time and
- the data in the array are separated in computer memory by the same distance, which means that inserting an item inside the array requires shifting other data in this array.

This limitation can be overcome by using ***linked structures***. A linked structure is a collection of nodes storing data and links to other nodes. In this way, nodes can be located anywhere in memory, and passing from one node of the linked structure to another is accomplished by storing the addresses of other nodes in the linked structure. Although linked structures can be implemented in a variety of ways, the most flexible implementation is by using *pointers*.

Singly Linked Lists

If a node contains a data member that is a pointer to another node, then many nodes can be strung together using only one variable to access the entire sequence of nodes. Such a sequence of nodes is the most frequently used implementation of a ***linked list***, which is a data structure composed of nodes, each node holding some information and a pointer to another node in the list. If a node has a link only to its successor in this sequence, the list is called a ***singly linked list***.

Each node in the list is an instance of the following class definition:



```
class Node
{
  int data;
  Node next;
  public Node()
  {
    data = 0 ;
    next = null;
  }
  public Node(int data)
  {
    this.data = data;
    this.next = null;
  }
}
```

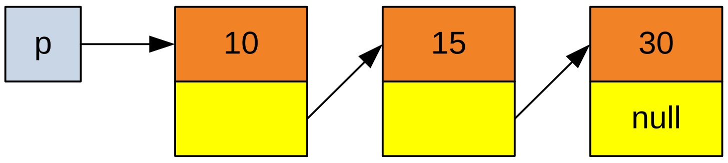
A node includes two data members: *data* and *next*. The *data* member is used to store information, and this member is important to the user. The *next* member is used to link nodes to form a linked list. It is an auxiliary data member used to maintain the linked list. It is indispensable for implementation of the linked list, but less important (if at all) from the user's perspective. Note that *Node* is defined in terms of itself because one data member, *next*, is a pointer to a node of the same type that is just being defined. Objects that include such a data member are called ***self-referential objects***.

The definition of a node also includes two constructors:

- the first constructor initializes the *next* pointer to null and leaves the value of *data* unspecified.
- the second constructor takes two arguments: one to initialize the *data* member and another to initialize the *next* member. The second constructor also covers the case when

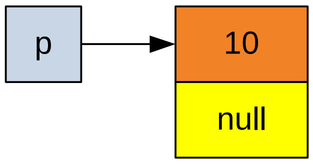
only one numerical argument is supplied by the user. In this case, *data* is initialized to the argument and *next* to **null**.

Let us create the next linked list:



One way to create this three-node linked list is to first generate the node for number 10, then the node for 15, and finally the node for 30. Each node has to be initialized properly and incorporated into the list.
Create the first node on the list and make the variable *p* a pointer to this node:

```
Node p = new Node(10);
```



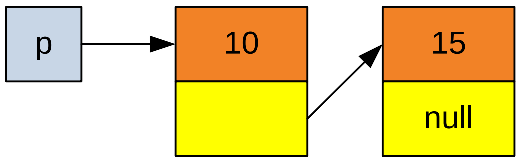
This done in four steps:

- Create a new *Node*;
- The *data* member of this node is set to 10;
- the node's *next* member is set to **null**;
- make *p* a pointer to the newly created node. This pointer is the *address* of the node, and it is shown as an arrow from the variable *p* to the new node.

The second node is created with the assignment:

```
p.next = new Node(15);
```

Here `p.next` is the *next* member of the node pointed to by *p*.



The linked list is now extended by adding a third node with the assignment

```
p.next.next = new Node(30);
```

Here `p.next.next` is the *next* member of the second node. This cumbersome notation has to be used because the list is accessible only through the variable *p*.

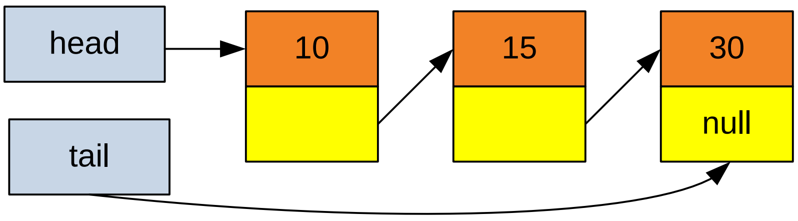
Our linked list example illustrates a certain inconvenience in using references: the longer the linked list, the longer the chain of *next*s to access the nodes at the end of the list. In this example, `p.next.next.next` allows us to access the *next* member of the 3rd node on the list. But what if it were the 103rd or, worse, the 1,003rd node on the list? Typing 1,003 *next*s, as in *p*`.next. . . . .next`, would be daunting. If we missed one *next* in this chain, then a wrong assignment is made. Also, the flexibility of using linked lists is diminished. Therefore, other ways of accessing nodes in linked lists are needed.

```
import java.util.*;

class Node
{
  int data;
  Node next;
  public Node()
  {
    data = 0;
    next = null;
  }
  public Node(int data)
  {
    this.data = data;
    this.next = null;
  }
}

public class Main
{
  public static void main(String[] args)
  {
    Node p = new Node(10);
    p.next = new Node(15);
    p.next.next = new Node(30);
    System.out.println(p.data + " " + p.next.data + " " + p.next.next.data);
  }
}
```

C++ implementation:

```
  #include <stdio.h>

  class ListNode
  {
  public:
    int data;
    ListNode *next;
    ListNode(int data) : data(data), next(NULL) {}
  };

  int main(void)
  {
    ListNode *temp = new ListNode(10);
    temp->next = new ListNode(15);
    temp->next->next = new ListNode(30);
    printf("%d %d %d\n", temp->data, temp->next->data, temp->next->next-
>data);
    return 0;
  }
```

One way is always to keep two pointers to the linked list: one to the first node and one to the last:



The singly linked list implementation uses two classes: one class, **Node**, for nodes of the list, and another, **LinkedList**, for access to the list. The class **LinkedList** defines two data members, *head* and *tail*, which are pointers to the first and the last nodes of a list. Method `Empty()` checks if the list is empty:

```
class LinkedList
{
  Node head, tail;
  public LinkedList()
  {
    head = null;
    tail = null;
  }
  public boolean Empty()
  {
    return head == null;
  }
}
```

Besides the *head* and *tail* members, the class **LinkedList** also defines member functions that allow us to manipulate the lists. We now look more closely at some basic operations on linked lists.

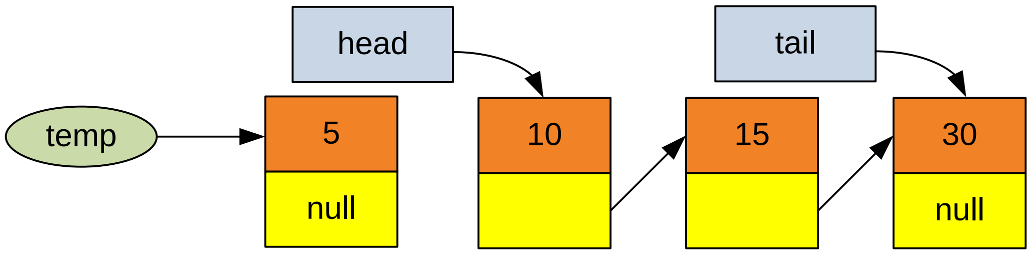
The list is declared with the statement:

```
LinkedList list = new LinkedList();
```

Insert an element to the start of Linked List
Adding a node at the ***beginning*** of a linked list is performed in three steps:

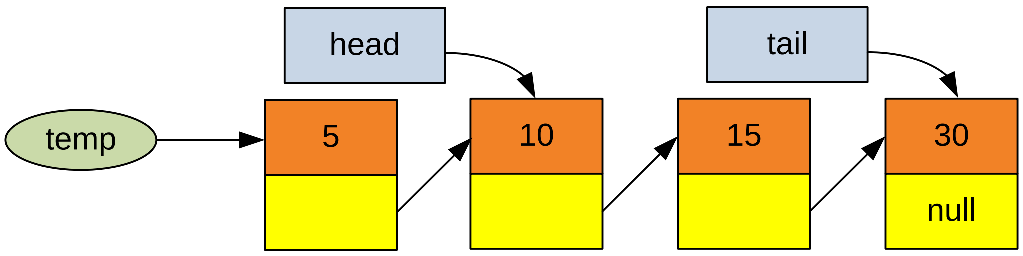
1. An empty node *temp* is created. The node's info member is initialized to a particular integer.

```
Node temp = new Node(5);
```



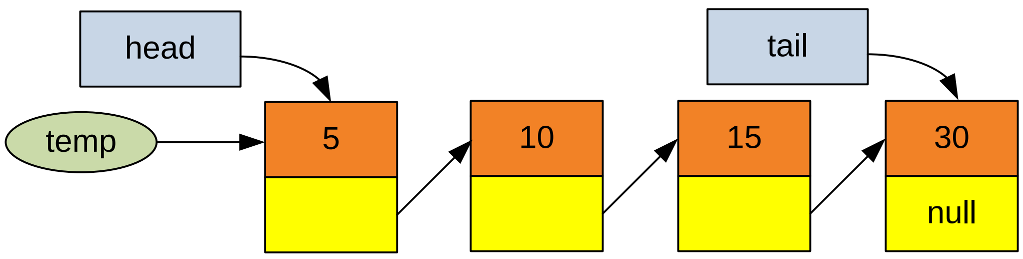
2. Because the node is being included at the front of the list, the *next* member becomes a pointer to the first node on the list; that is, the current value of head.

```
temp.next = head;
```



3. The new node precedes all the nodes in the list, but this fact has to be reflected in the value of *head*; otherwise, the new node is not accessible. Therefore, *head* is updated to become the pointer to the new node.

```
head = temp;
```



Three steps are executed by the member function ***addFirst***():

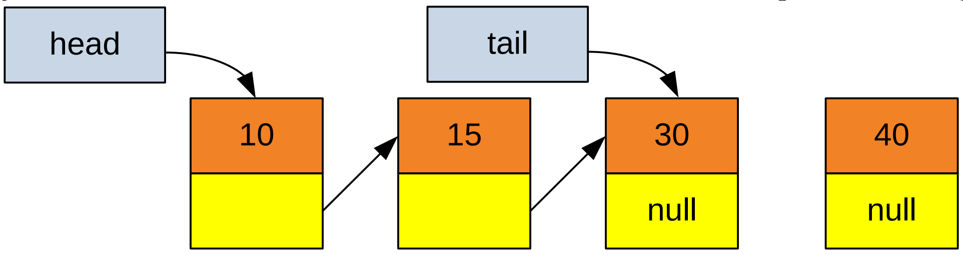
```
public void addFirst(int val)
{
  if (tail == null) // list is empty
    head = tail = new Node(val);
  else
  {
    Node temp = new Node(val);
    temp.next = head;
    head = temp;
  }
}
```

The member function ***addFirst*** () singles out one special case, namely, inserting a new node in an empty linked list. In an empty linked list, both *head* and *tail* are null; therefore, both become references to the only node of the new list. When inserting in a nonempty list, only *head* needs to be updated.

Insert an element to the end of Linked List

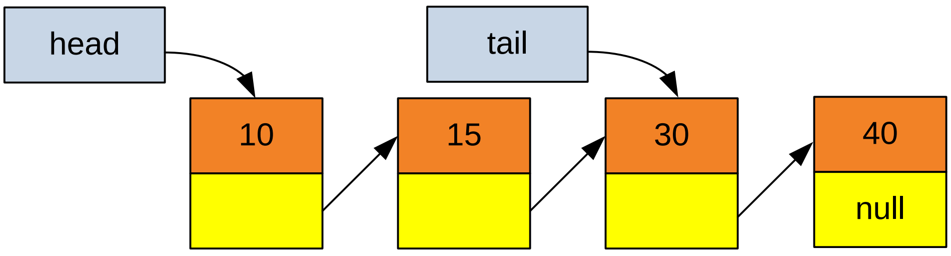
The process of adding a new node ***to the end*** of the list has three steps.

1. An empty node is created. The node's info member is initialized to a particular integer.



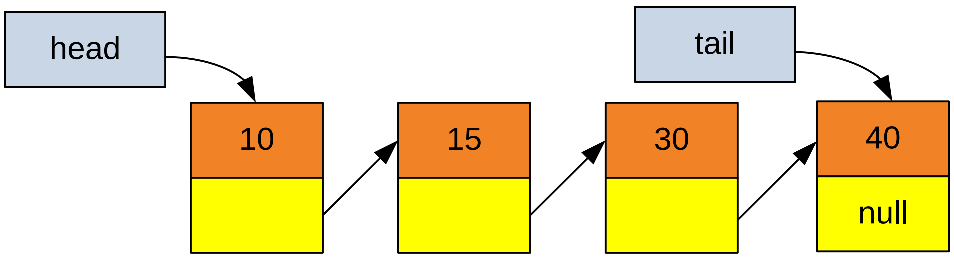
2. The node is now included in the list by making the *next* member of the last node of the list a pointer to the newly created node.

```
tail.next = new Node(40);
```



3. The new node follows all the nodes of the list, but this fact has to be reflected in the value of *tail*, which now becomes the pointer to the new node.

```
tail = tail.next;
```



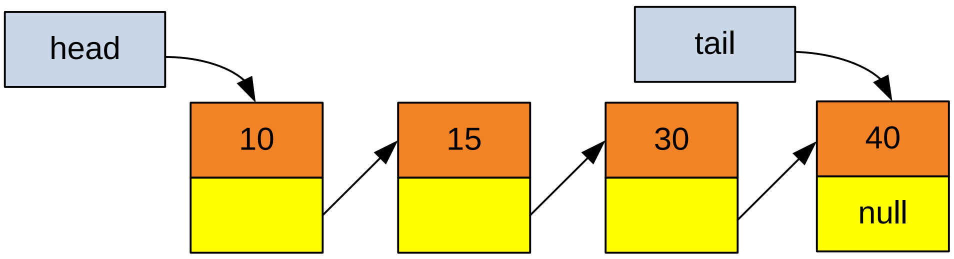
```
public void addLast(int val)
{
  if (tail != null) // list is not empty
  {
    tail.next = new Node(val);
    tail = tail.next;
  }
  else head = tail = new Node(val);
}
```

All these steps are executed in the if clause of ***addLast***(). The else clause of this function is executed only if the linked list is empty.

Delete an element from the start of Linked List

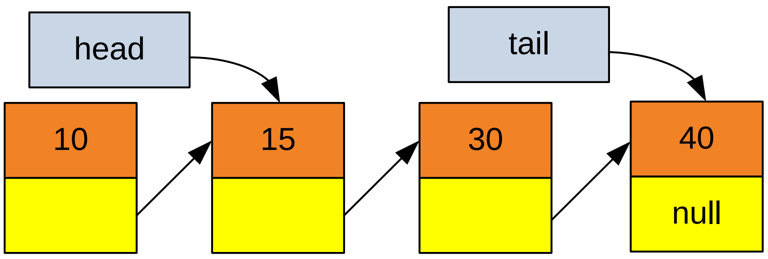
Function ***removeFirst*** returns `true` is deletion is successful, otherwise `false`. There are two special cases to consider:

- One case is when we attempt to remove a node from an empty linked list. If such an attempt is made, the program is very likely to crash, which we don’t want to happen. So in this case we do nothing, but simply return `false`.
- The second special case is when the list has only one node to be removed. In this case, the list becomes empty, which requires setting *tail* and *head* to **null**.



Move the *head* pointer one position forward.

```
head = head.next;
```



```
public boolean removeFirst()
{
  if (Empty()) return false;
  if (head == tail) // only one node in a list
    head = tail = null;
  else head = head.next;
  return true;
}
```

Delete an element from the end of Linked List

Now consider the process of deleting a node from the end of the list. It is implemented as the member function ***removeLast***(). The problem is that after removing a node, *tail* should refer to the new tail of the list; that is, *tail* has to be moved backward by one node. But moving backward is impossible because there is no direct link from the last node to its predecessor. Hence, this predecessor has to be found by searching from the beginning of the list and stopping right before tail. This is accomplished with a temporary variable *temp* that scans the list within the **for** loop. The variable *temp* is initialized to the *head* of the list, and then in each iteration of the loop it is advanced to the next node.

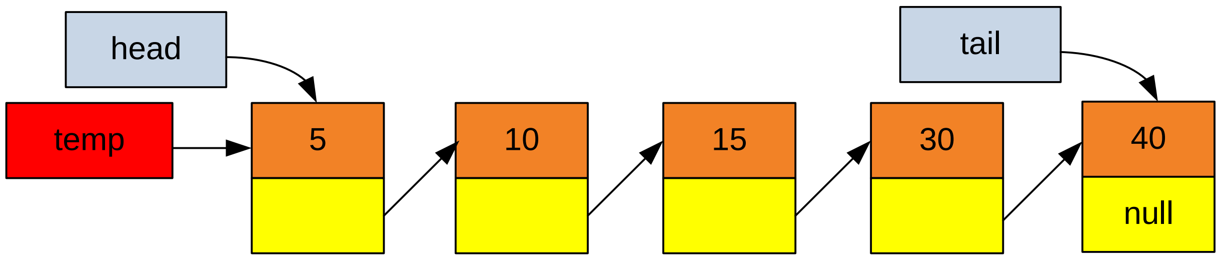
In removing the last node, the two special cases are the same as in ***removeFirst***():

- If the list is empty, then nothing can be removed;
- When a single-node list becomes empty after removing its only node, which also requires setting *head* and *tail* to **null**.

```
public boolean removeLast()
{
  if (Empty()) return false;
  if (head == tail) // only one node in a list
  {
    head = tail = null;
  }
  else // if more than one node in the list
  {
    Node temp = head;
    // find the predecessor of  tail
    while(temp.next != tail) temp = temp.next;
    tail = temp; // the predecessor of  tail becomes tail
    tail.next = null;
  }
  return true;
}
```

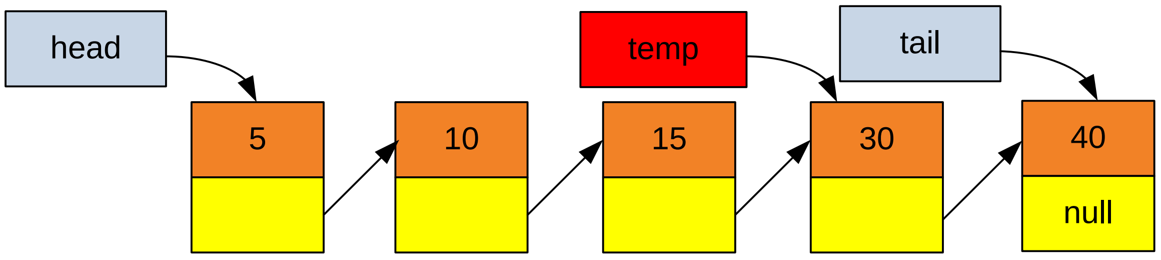
Consider the list, where *temp* first refers to the head node holding number 5.

```
temp = head;
```



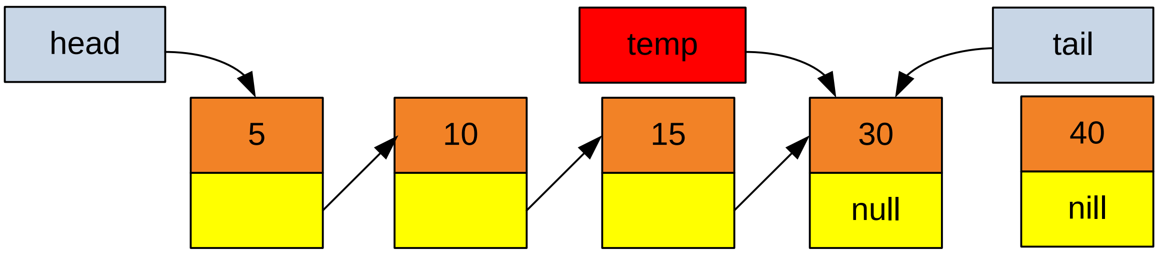
Move *temp* forward until it will point to the node next to the last.

```
while(temp.next != tail) temp = temp.next;
```



The last node is deleted.

```
tail = temp; tail.next = null;
```



The most time-consuming part of deleteFromTail() is finding the next to last node performed by the ***while*** loop. It is clear that the loop performs $n - 1$ iterations in a list of n nodes, which is the main reason this member function takes O(n) time to delete the last node.

Java implementation:

```
import java.util.*;

class ListNode
{
  int data;
  ListNode next;
  public ListNode(int data)
  {
    this.data = data;
    this.next = null;
  }
}

class LinkedList
{
  ListNode head, tail;
  public LinkedList()
  {
    head = null;
    tail = null;
  }

  public boolean Empty()
  {
    return head == null;
  }

  public void addFirst(int val)
  {
    if (tail == null) // list is empty
      head = tail = new ListNode(val);
    else
    {
      ListNode temp = new ListNode(val);
      temp.next = head;
      head = temp;
    }
  }

  public void addLast(int val)
```

```java
  {
    if (tail != null) // list is not empty
    {
      tail.next = new ListNode(val);
      tail = tail.next;
    }
    else head = tail = new ListNode(val);
  }

  public boolean removeFirst()
  {
    if (Empty()) return false;
    if (head == tail) // only one node in a list
      head = tail = null;
    else head = head.next;
    return true;
  }

  public boolean removeLast()
  {
    if (Empty()) return false;
    if (head == tail) // only one node in a list
    {
      head = tail = null;
    }
    else // if more than one node in the list
    {
      ListNode temp = head;
      // find the predecessor of  tail
      while(temp.next != tail) temp = temp.next;
      tail = temp; // the predecessor of  tail becomes tail
      tail.next = null;
    }
    return true;
  }

  public void Print()
  {
    ListNode head = this.head;
    while(head != null)
    {
      System.out.print(head.data + " ");
      head = head.next;
    }
    System.out.println();
  }
}

public class Main
{
  public static void main(String[] args)
  {
     LinkedList list = new LinkedList();
     list.addFirst(10);    list.addFirst(15);
     list.removeLast();    list.addFirst(20); // 20 15
     list.Print();

     list.addLast(30);    list.addLast(35);
    list.addFirst(77);   list.addFirst(99); // 99 77 20 15 30 35
    list.Print();

    list.removeFirst(); list.removeFirst();
    list.removeFirst(); list.removeLast();  // 15 30
    list.Print();
  }
}
```

C++ implementation:

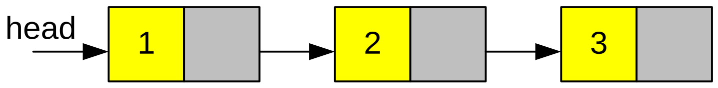
```
#include <stdio.h>

class ListNode
{
public:
  int data;
  ListNode *next;
  ListNode(int data) : data(data), next(NULL) {}
};

class LinkedList
{
public:
  ListNode *head, *tail;
  LinkedList()
  {
    head = NULL;
    tail = NULL;
  }

  bool Empty()
  {
    return head == NULL;
  }

  void addFirst(int val)
  {
    if (tail == NULL) // list is empty
    head = tail = new ListNode(val);
    else
    {
      ListNode *temp = new ListNode(val);
      temp->next = head;
      head = temp;
    }
  }

  void addLast(int val)
  {
    if (tail != NULL) // list is not empty
    {
      tail->next = new ListNode(val);
      tail = tail->next;
    }
    else head = tail = new ListNode(val);
  }
  bool removeFirst()
  {
    if (Empty()) return false;
    if (head == tail) // only one node in a list
      head = tail = NULL;
    else head = head->next;
    return true;
  }

  bool removeLast()
  {
    if (Empty()) return false;
    if (head == tail) // only one node in a list
    {
      head = tail = NULL;
    }
    else // if more than one node in the list
    {
      ListNode *temp = head;
      // find the predecessor of  tail
      while (temp->next != tail) temp = temp->next;
        tail = temp; // the predecessor of  tail becomes tail
      tail->next = NULL;
    }
    return true;
  }

  void Print()
  {
    ListNode *head = this->head;
    while (head != NULL)
    {
      printf("%d ", head->data);
      head = head->next;
    }
    printf("\n");
  }
};

ListNode *temp;

int main(void)
{
  LinkedList *list = new LinkedList();
  if (list->Empty()) printf("Empty\n");
  list->addFirst(10);  list->addFirst(15);
  list->removeLast(); list->addFirst(20);
  list->Print(); // 20 15

  list->addLast(30);   list->addLast(35);
  list->addFirst(77);  list->addFirst(99);
  list->Print(); // 99 77 20 15 30 35

  list->removeFirst(); list->removeFirst();
  list->removeFirst(); list->removeLast();
  list->Print(); // 15 30
  return 0;
}
```

E-OLYMP [9898. LinkedList Sum] Given a linked list, find the sum of its elements.

► You need to implement a function

- `int sum(ListNode head) // Java`
- `int sum(ListNode *head) // C++`

that finds the sum of linked list elements.



The sum of elements of a linked list is 6.

Run the *head* from the start to the end of the list, node after node and sum up the *val* numbers in the variable *res*.

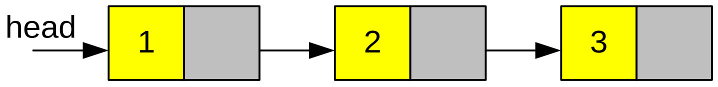
```
// C++ implementation
int sum(ListNode *head)
{
  int res = 0;
  while (head != NULL)
  {
    res += head->val;
    head = head->next;
  }
  return res;
}

// Java implementation
int sum(ListNode head)
{
  int res = 0;
  while(head != null)
  {
    res += head.val;
    head = head.next;
  }
  return res;
}
```

E-OLYMP [9899. LinkedList Length] Given a linked list, find the ***length*** of the linked list.

► You need to implement a function

- `int length(ListNode head) // Java`
- `int length(ListNode *head) // C++`

that finds the length of linked list.



The length of a linked list is 3.

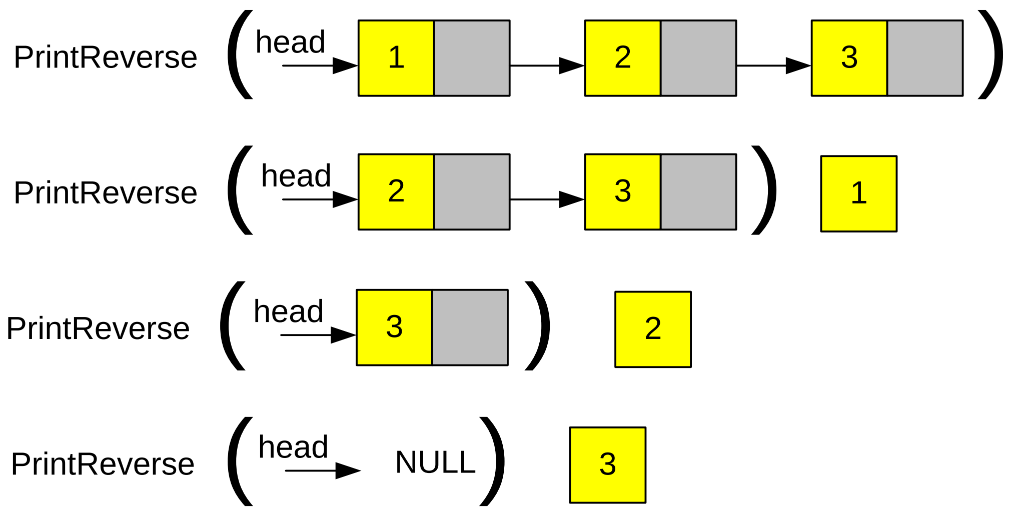
Iterate the linked list from *head* to *tail* and count the number of elements.

E-OLYMP [10041. Print in reverse order] Given a linked list. Print its elements in the reverse order.

► Implement the **PrintReverse** function as follows: first, print the elements of the tail of the linked list in reverse order, and then print the value of the head.

```
PrintReverse(ListNode head)
    if head == NULL, return NULL;
    PrintReverse(head->next) ;
    print head->val;
```

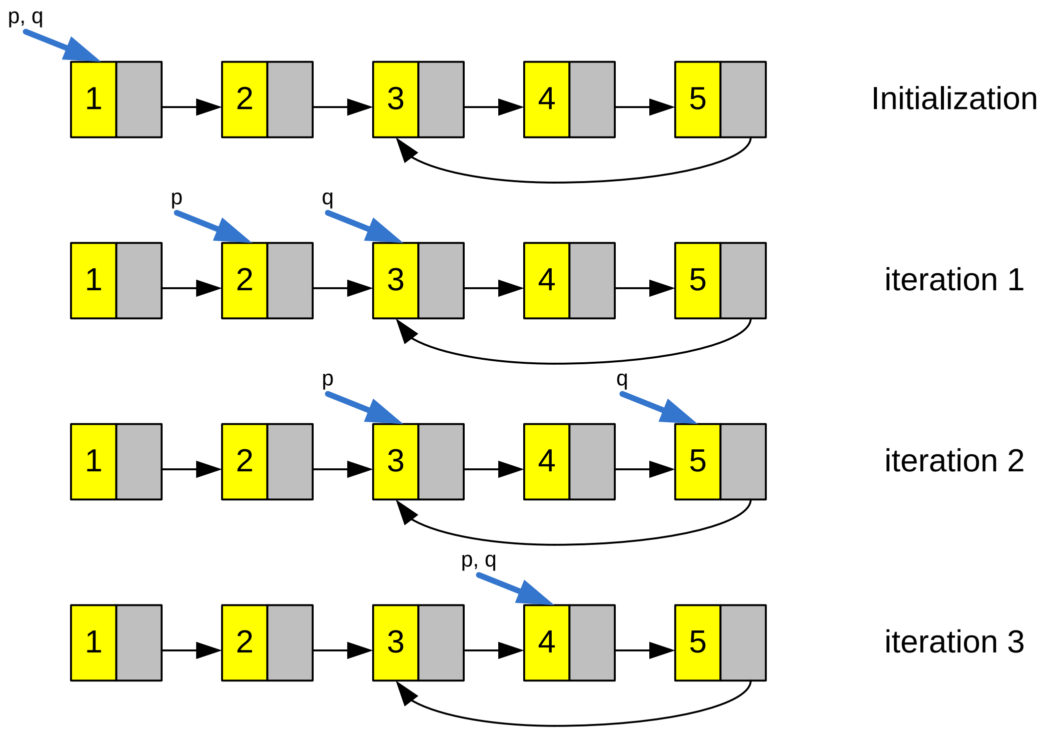
Consider how to print the list given in example in the reverse order.



E-OLYMP 10042. LinkedList Cycle Given a linked list. Does it contain a cycle?

► We'll use two pointers *p* and *q*, moving them according to the principle of ***baby step giant step***. First, assign them to the start of the list. Next, iteratively move the first pointer *p* forward one position in the list, and the second pointer *q* move two positions forward in the list. We stop if:

- one of the pointers (this will be the pointer *q*) has reached the end of the list. Then the list has no cycle.
- both pointers point to one element. This means that the fast pointer *q* caught the slow pointer *p*, and the list contains a cycle.



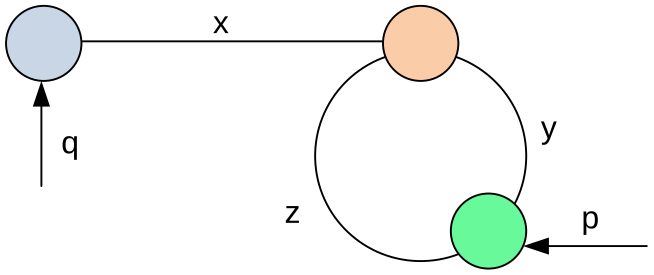
At the third iteration, pointer *q* caught *p*, both pointers point to a vertex with a value of 4. Cycle exists.

```
// C++ implementation
int hasCycle(ListNode *head)
{
  if (head == NULL) return 0;
  ListNode* p = head;
  ListNode* q = head;
  while (q->next && q->next->next)
  {
    p = p->next;
    q = q->next->next;
    if (p == q) return 1;
  }
  return 0;
}
```

```
// Java implementation
int hasCycle(ListNode head)
{
  if (head == null) return 0;
  ListNode p = head;
  ListNode q = head;
  while (q.next != null && q.next.next != null)
  {
    p = p.next;
    q = q.next.next;
    if (p == q) return 1;
  }
  return 0;
}
```

E-OLYMP 10043. LinkedList Cycle starting point Given a linked list. Return pointer to the node where cycle starts. If there is no cycle, return **null**.

► Using two pointers (*slow* and *fast*), we determine whether the linked list contains a loop. Let the list contains a loop, and *p* points to the vertex where the fast pointer meets with the slow one. Set *q* to the start of the list.



Let the distance from the start to the vertex where the cycle starts is *x*. The distance from the start of the loop to the meeting point of two pointers is *y*. The distance from the vertex pointed to by *p* to the beginning of the cycle along the list is *z*.

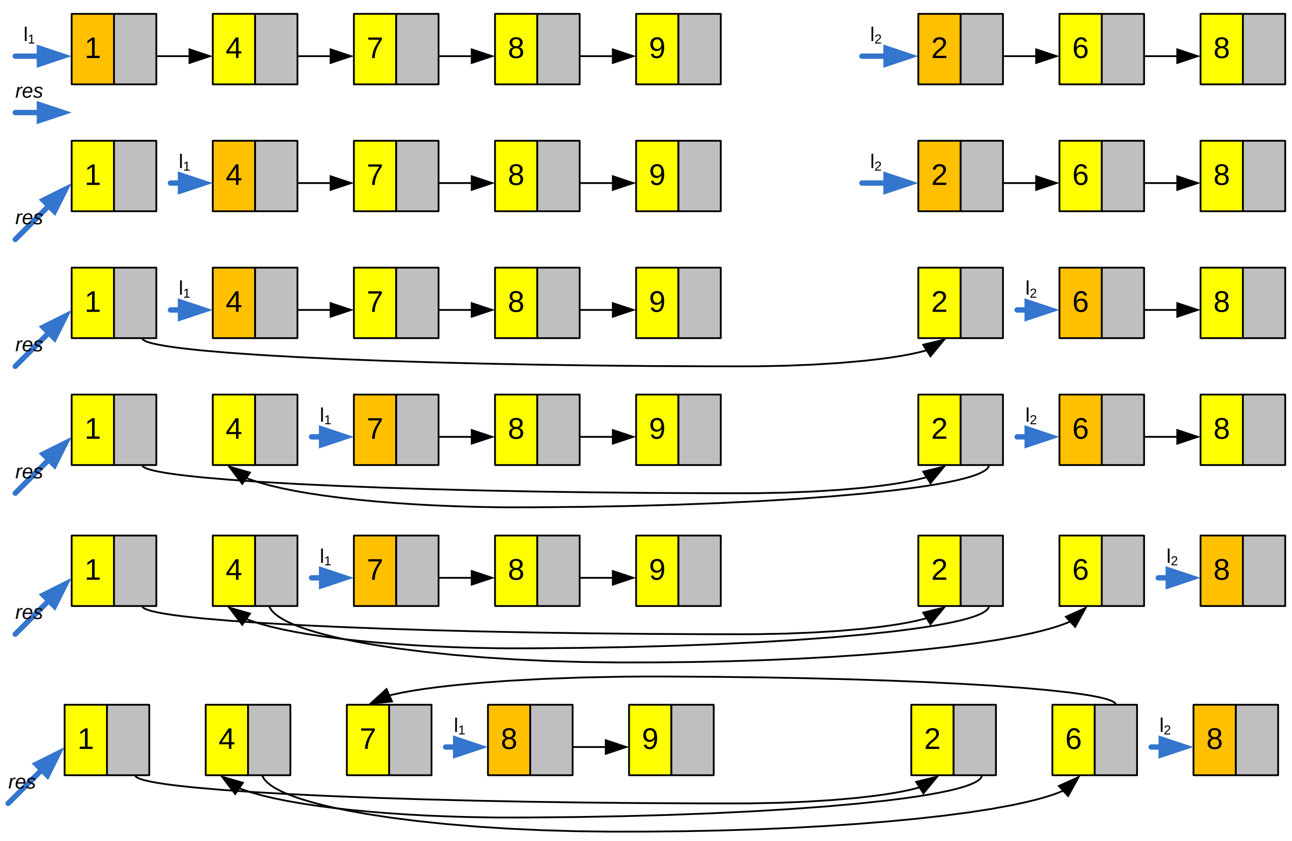
Suppose the pointers have met so that the fast pointer has only gone one round of the cycle. That is, the slow one went the path $x + y$, and the fast one $x + y + z + y$. Considering that a fast pointer is twice as fast as a slow one, we get an equality:

$$2\,(x + y) = x + y + z + y \text{ or } x = z$$

This means that now it is enough to iteratively move the pointers *p* and *q* step by step until they become equal (will point to the same memory location). Moreover, this will definitely happen at the node, which is the beginning of the cycle.

E-OLYMP 10044. LinkedList Merge Merge two sorted linked lists and return it as a new list. The new list should be made by splicing together the nodes of the first two lists.

► Let *res* be the pointer to the resulting linked list. To the end of *res* we'll assign the head of either the first or the second list (the one that is smaller). If the first list ends earlier, then we assign to *res* what remains from the second list. If the second list ends earlier, then we assign to *res* what remains from the first list.

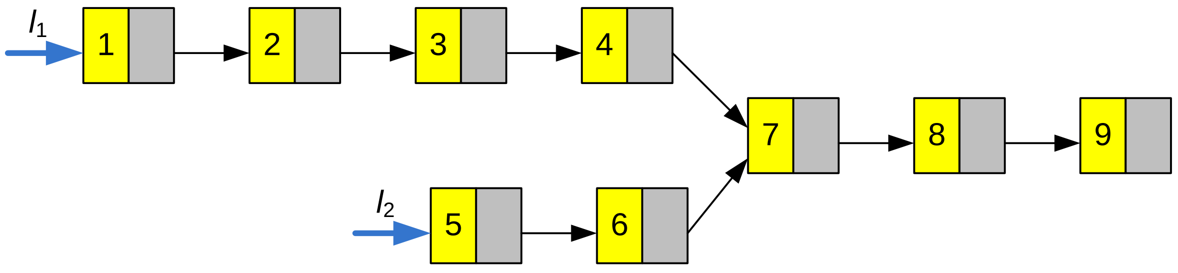


E-OLYMP 10047. LinkedList Intersection Find the point of intersection of two singly linked lists. Return the pointer to the node where the intersection of two lists starts.

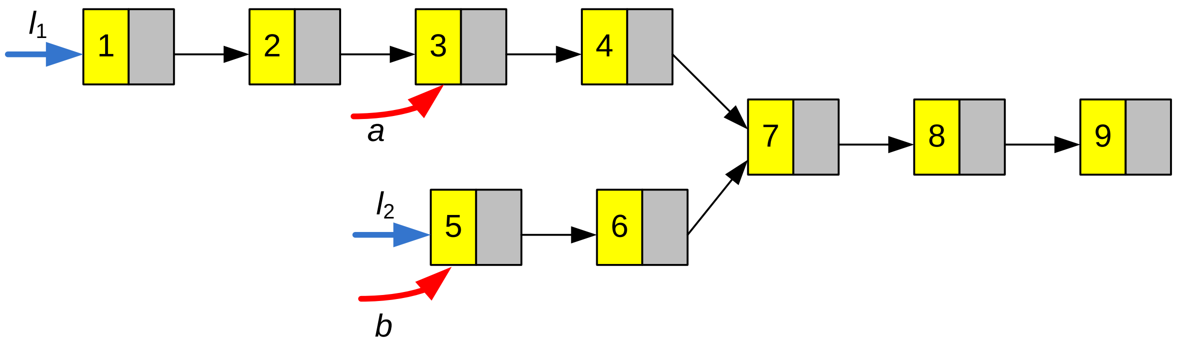
► Let's assume that input lists have the same length. Set up to each of its heads a pointer. Move consecutively both pointers one position to the right until they point to the same memory location.

However, in our case, the lengths of the lists may be different. Let *lenA* and *lenB* be the lengths of the lists (they can be computed in O(*n*)). Let's set up two pointers to the heads of the lists. The pointer, set on the head of a longer list, will be moved forward $| lenA - lenB |$ positions forward. Then solve the problem as if the lists have the same length.

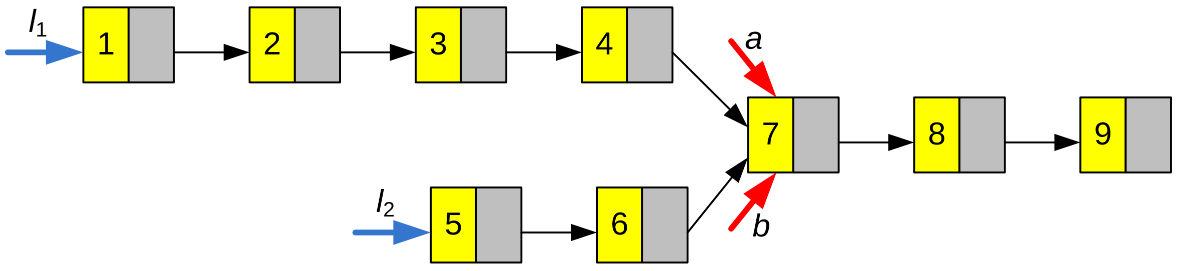
Example. The length of the list l_1 is *lenA* = 7. The length of the list l_2 is *lenB* = 5.



Set up the pointers $a = l_1$, $b = l_2$. Move the pointer a $|7 - 5| = 2$ positions forward.



Step by step move the pointers *a* and *b* one position forward until they become equal. Once *a* = *b*, the pointers will point to the intersection point.



Leetcode interview questions: https://leetcode.com/problemset/all/

21. Merge Two Sorted Lists
https://leetcode.com/problems/merge-two-sorted-lists/

141. Linked List Cycle
https://leetcode.com/problems/linked-list-cycle/

142. Linked List Cycle II
https://leetcode.com/problems/linked-list-cycle-ii/

160. Intersection of Two Linked Lists
https://leetcode.com/problems/intersection-of-two-linked-lists/