Singly (Simply) Linked Lists

List

List is a set of items or elements, list can be:

- ***Non-linked List***: such type is implemented using arrays, the locations of non-linked don't need to link because they are sequential in RAM.
- ***Linked List***: such type needs pointers for implementing it, pointer does as linker between the non-sequential locations.

Linked Lists

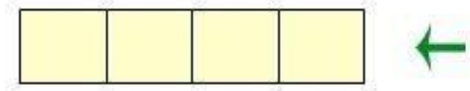
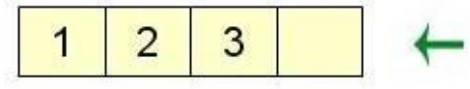
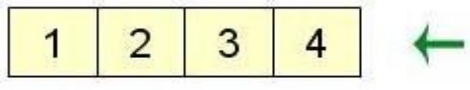
In order to avoid the linear cost of insertion and deletion, we need to ensure that the list is not stored contiguously, since otherwise entire parts of the list will need to be moved. A linked list, in its simplest form, is a collection of **nodes** that **together form a linear ordering**.

Linked Lists are a very common way of storing arrays of data. The major benefit of linked lists is that you do not specify a fixed size for your list. The more elements you add to the chain, the bigger the chain gets.

The linked list consists of a series of nodes, which are not necessarily adjacent in memory. Each node contains the element and a link to a node containing its successor. We call this the next link. The last cell's next link references null.

There is **more than one type of a linked list**, we'll stick to **singly linked lists** (the simplest one). If for example you want a **doubly linked list** instead, very few simple modifications will give you what you're looking for. Many data structures (e.g. Stacks, Queues, Binary Trees) are often implemented using the concept of linked lists.

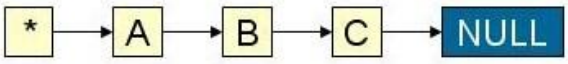
There's a diagram helps you to realize the main **disadvantage of arrays** but not Linked Lists

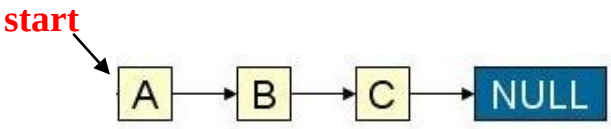
	Create an empty integer array.
	Fill it partially with some data.The array component without a number indicates allocated but unused space. This is space you could have used for something better.
	Add another element. We now have a full array to which we can not add any more elements. We can delete or replace, but we can not add.
	If the array is full, you can not add more elements, because arrays have a fixed size. Linked Lists do not.

Another drawback of arrays is that if you delete an element from the **middle** and want no holes in your array (e.g. (1, 2, 4, null) instead of (1, 2, null, 4)), you will need to shift everything after the deleted element down. If you're trying to add an element somewhere other than the very end of an array, you will need to **shift** some elements towards the end by one to make room for the new element, and if you're writing an application which needs to perform well and needs to do these operations often, you should consider using **Linked Lists** instead.

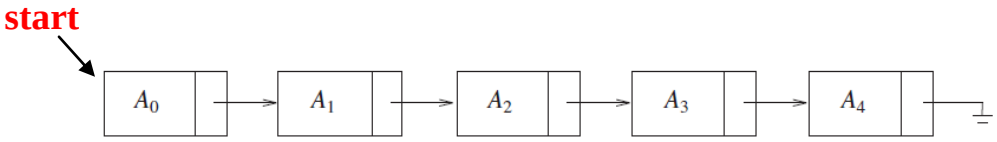
In **Singly Linked List** :

Root node links one way through all the nodes. Last node links to null.

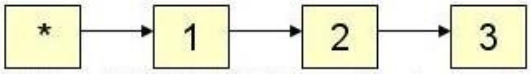
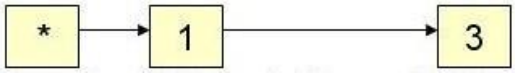
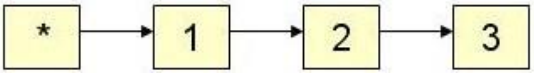




start is a pointer to the first node of linked list.



This should help you understand Linked Lists:

	This is an empty linked list look like. The ***** is an empty node which has its next-node reference set to the first node in the list. Since we don't have a first node, its next –node is a null pointer
	This is a Linked List with **three nodess**. Each node points to the **next node** in the chain. As mentioned above, ***** is an empty node with a reference to the first node. **[3]** is the last node in the chain with **next ==null**
	Here we have delete node **[2]**, so node **[1]** (previosly pointing to **[2]**) now points to **[3]**. If we didn't change the refernce, node **[3]** and any nodes behind it would have no references from your program and get lost. Thus you make sure update the refernces.
	For whatever reasons, we've decided to add node **[2]** between nodes **[1]** and **[3]**. The reference from **[1]** is set to **[2]**, and the refence from **[2]** is set to the old reference of **[1]**, which is **[3]**

Pointers and References

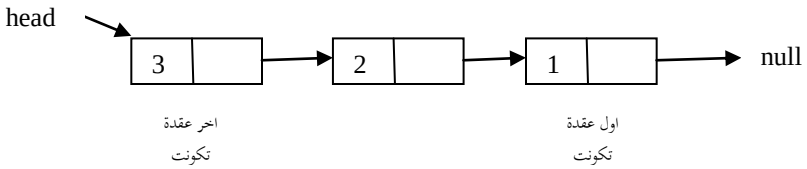
Java does not have an explicit pointer type. Instead of pointers, all **references to objects** (**including variable assignments, arguments passed into methods and array elements**) are accomplished by using implicit references. Reference semantics also enable structures such as linked lists to be created easily in Java without explicit pointers; just create a linked list node with variables that point to the next and the previous node.

```
public class link
  {
      int value;          // value of element
      link next;          // reference to next

      public link(int n, link ln)          // constructor
        {    value = n;
             next = ln;
         }
```

```
public static void main()
  {
      link head = null;              // initialize list head
      for (int i = 1; i <= 10; i++)
          head = new link(i, head);          // add some entries to list
```

في هذا الحل يتم بناء القائمة الموصولة من الاخير، وتضاف كل عقدة جديدة ليسار العقدة السابقة.




```
      // dump out entries  (printing)
      link p = head;
      while (p != null)
        {
            System.out.println(p.value);
            p = p.next;
         }
  }
}
```

ستطبع العقد بالشكل

3 2 1 →

- A line like:

  link next;

Does not actually declare an object of class link, but rather لكن بالاحرى a reference to an object.

- The line:

  head = new link(i, head);

Creates a new element for the list (a **node**, it is an **object**), sets its value, and then sets the object reference in "**next**" to point at the old list head (this approach means that the last element inserted will be at the head of the list).

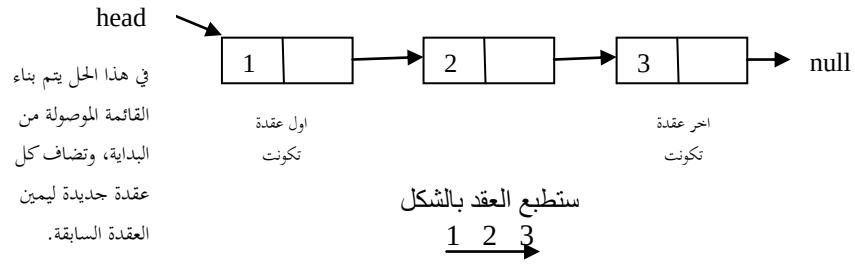
- Saying:

p = p.next;

Simply picks up the "next" reference from the current node and makes it the current element being worked on.

There is another solution to build a linked list as the following steps:

```
Read  n       //  the no of nodes
i=1
link  head = new  link (i,null)   //  build the first node that pointed by head pointer
node = head                       //  point the first node with another pointer (node)
for i=2     i<=n     i++
  {    p = new link (i, null)
       node.next = p
       node = p
  }
```




```
p = new link (i, null)
node.next = p
node = p
```

يكافئ

```
node.next = new link (i, null)
node = node.next
```

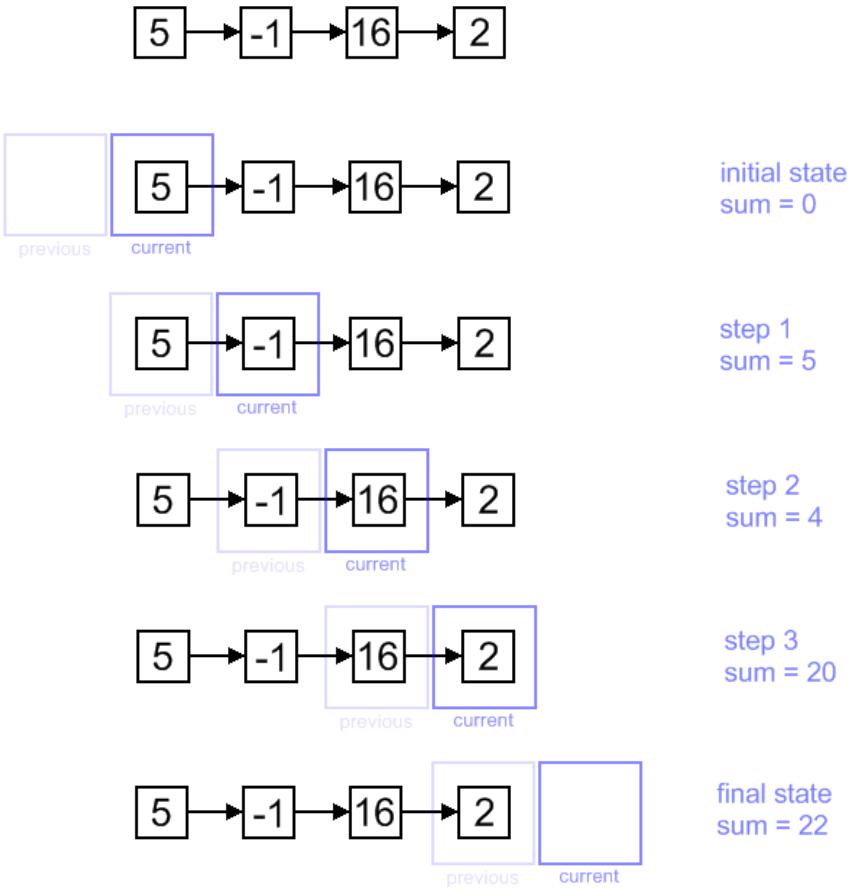
Traversal algorithm

Beginning from the head,

1. check, if the end of a list hasn't been reached yet;
2. do some actions with the current node, which is specific for particular algorithm;
3. current node becomes previous and next node becomes current. Go to the step 1.

Example

As for example, let us see an example of summing up values in a singly-linked list.



For some algorithms tracking the previous node is essential, but for some, like an example, it's unnecessary. We show a common case here and concrete algorithm can be adjusted to meet its individual requirements.