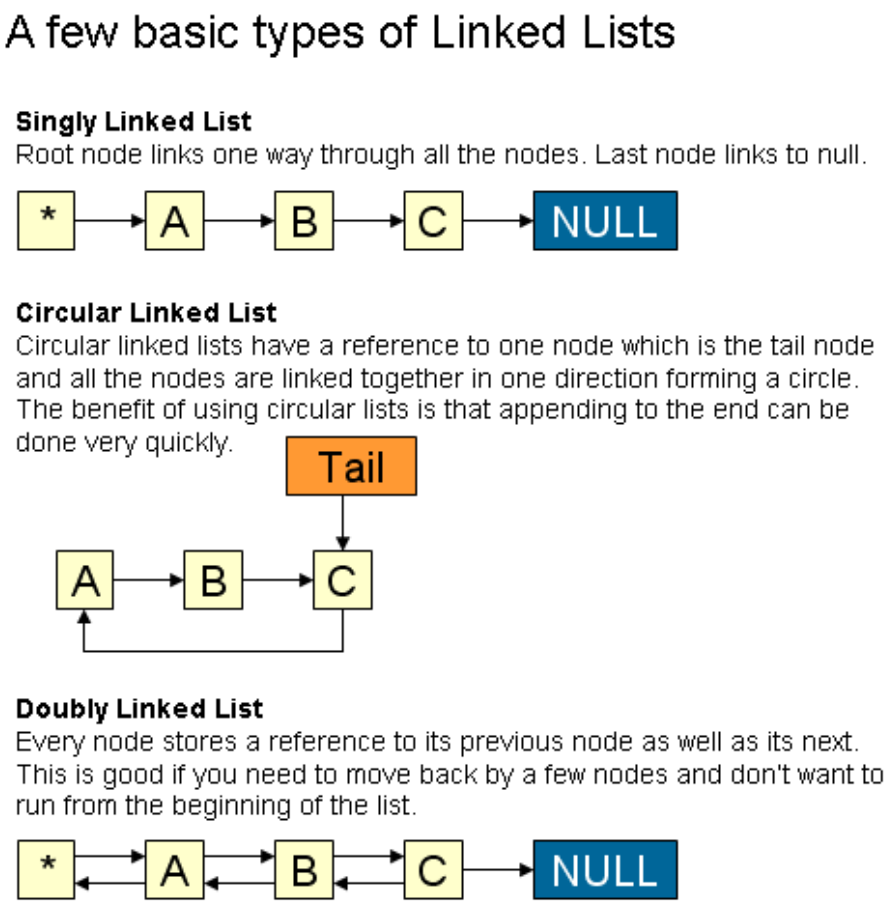
menu
- Articles

featured articles
- QuickSort Tutorial
- Linked Lists Tutorial

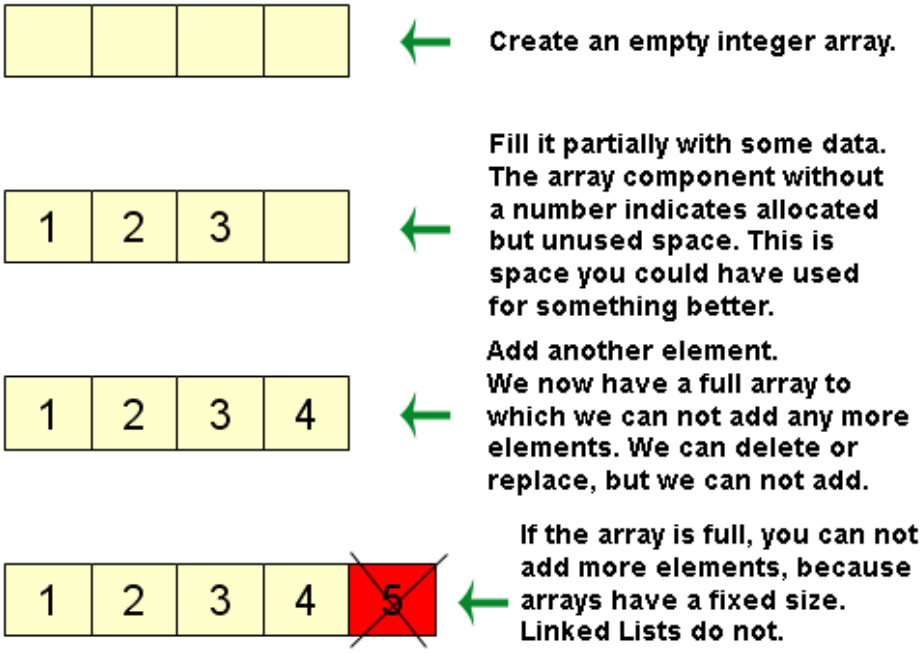
Linked Lists

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.

There is more than one type of a linked list, although for the purpose of this tutorial, 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. Some different types of linked lists are shown below:

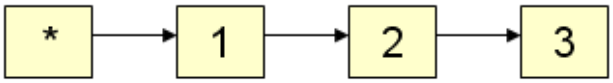


Here's a diagram to help you realize the main disadvantage of arrays but not Linked Lists (there are certain advantages of using arrays over linked lists, but they are not covered in this article):



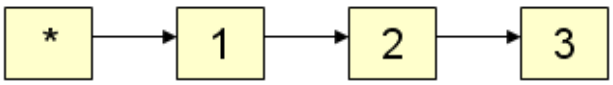
Another drawback of arrays is that if you delete an element from the middle and want no holes in your array (e.g. (1, 2, 4, null) instead of (1, 2, null, 4)), you will need to shift everything after the deleted element down in O(n) time. If you're trying to add an element somewhere other than the very end of an array, you will need to shift some elements towards the end by one (also O(n) time) 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. This should help you understand Linked Lists:

This is what an empty Linked List looks like. The * is an empty node (each element in a Linked List is called a 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 nodes. 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 deleted node [2], so node [1] (previously pointing to [2]) now points to [3]. If we didn't change the reference, node [3] and any nodes behind it would have no references from your program and get lost. If this happens and you're using C or C++, you have a memory leak. If you're using Java, the nodes would get automatically garbage collected. Either way, make sure you update the references!



For whatever reasons, we've decided to add node [2] back between nodes [1] and [3]. The reference from [1] is set to [2], and the reference from [2] is set to the old reference of [1], which is [3].

My code of the LinkedList class follows. It is a commented oversimplification of the LinkedList class that's already built into Java. Please note that indexes start at 1 and not 0 (like in the Java LinkedList class), so the first element in the list has index 1.

```
blic class LinkedList

	// reference to the head node.
	private Node head;
	private int listCount;

	// LinkedList constructor
	public LinkedList()
	{
		// this is an empty list, so the reference to the head node
		// is set to a new node with no data
		head = new Node(null);
		listCount = 0;
	}

	public void add(Object data)
	// post: appends the specified element to the end of this list.
	{
		Node temp = new Node(data);
		Node current = head;
		// starting at the head node, crawl to the end of the list
		while(current.getNext() != null)
		{
			current = current.getNext();
		}
		// the last node's "next" reference set to our new node
		current.setNext(temp);
		listCount++;// increment the number of elements variable
	}

	public void add(Object data, int index)
	// post: inserts the specified element at the specified position in this list.
```

```
    {
            Node temp = new Node(data);
            Node current = head;
            // crawl to the requested index or the last element in the list,
            // whichever comes first
            for(int i = 1; i < index && current.getNext() != null; i++)
            {
```

Written by Pavel.