# Linked Lists

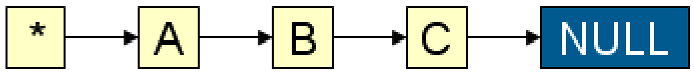
Linked Lists are a very common way of storing pieces of data. The major benefit of linked lists is that you can insert and delete records from anywhere in the list. You also do not need to specify a fixed size for your list. The more elements you add to the chain, the bigger the chain gets.

There is more than one type of linked list, although for our grade 12 class, we'll stick to singly linked lists (the simplest one). See below for the different types of linked lists.

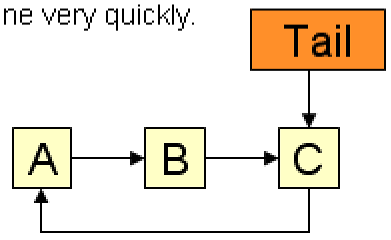
Many data structures (e.g. Stacks, Queues, Binary Trees) are often implemented using the concept of linked lists. Some different types of linked lists are shown below:

## A few basic types of Linked Lists

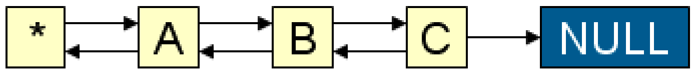
**Singly Linked List**
Root node links one way through all the nodes. Last node links to null.



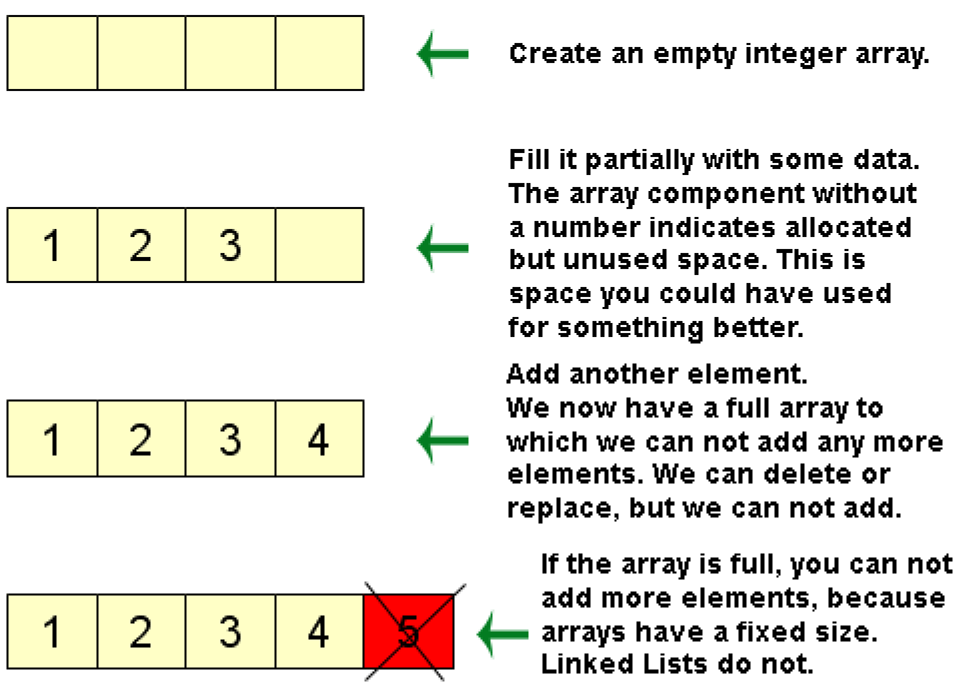
**Circular Linked List**
Circular linked lists have a reference to one node which is the tail node and all the nodes are linked together in one direction forming a circle. The benefit of using circular lists is that appending to the end can be done very quickly.



**Doubly Linked List**
Every node stores a reference to its previous node as well as its next. This is good if you need to move back by a few nodes and don't want to run from the beginning of the list.



Here's a diagram to help you realize the main advantage of using a Linked List over an array:


Create an empty integer array.
1 2 3
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.
1 2 3 4
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.
1 2 3 4 5
If the array is full, you can not add more elements, because arrays have a fixed size. Linked Lists do not.

Another drawback of using arrays is that if you delete an element from the middle and want no holes in your array, you will need to shift everything after the deleted element.

Also, if you're trying to add an element somewhere inside an array, you will need to shift elements after the insertion point towards the end to make room for the new element.

Individual elements in a Linked Lists are called NODES.

Here is what a Linked List looks like:

$$\begin{tikzcd}\boxed{*}\end{tikzcd}$$

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.

$$\begin{tikzcd}\boxed{*} \arrow[r] & \boxed{1} \arrow[r] & \boxed{2} \arrow[r] & \boxed{3}\end{tikzcd}$$

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.

$$\begin{tikzcd}\boxed{*} \arrow[r] & \boxed{1} \arrow[rr] & & \boxed{3}\end{tikzcd}$$

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!

$$\begin{tikzcd}\boxed{*} \arrow[r] & \boxed{1} \arrow[r] & \boxed{2} \arrow[r] & \boxed{3}\end{tikzcd}$$

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 ].

This implementation uses the Java API's LinkedList class.

```
/*
Simple Java LinkedList example...
This java example shows how to use the Java API LinkedList class
to create a Linked List and add some items to it.
*/

import java.util.LinkedList;

public class LinkedListExample {

  public static void main(String[] args) {

    //create an empty LinkedList object
    LinkedList mList = new LinkedList();

    /* Add nodes to the LinkedList object
     * with the data specified.
     */

    mList.add("every");           // index 0
    mList.add("day");             // index 1
    mList.add("I");               // index 2
    mList.add("eat");             // index 3
    mList.add("carrot");          // index 4

    /*
     * primitives (like integers) cannot be added
     * into a LinkedList object directly.
     * They must be converted to their corresponding
     * wrapper class.
     */

    System.out.println("LinkedList contains : " + mList);

    /* Insert a node into the list
     * between index 3 and 4.
     */

    mList.add(4, "a");
    System.out.println("LinkedList contains : " + mList);
    /* Remove nodes from the list
     * at index 4 and 5.
     */

    mList.remove(4);  // remove node at index 4
           // After node 4 is deleted,
           // the original node 5 has moved into node 4.
           // Therefore, to remove the original node 5,
           // delete node 4.
    mList.remove(4);
    System.out.println("LinkedList contains : " + mList);

  }
}
```

Some Linked List methods that are contained in the Java API's LinkedList class.

```java
/*
  Get a SubList from a LinkedList.
*/

import java.util.LinkedList;
import java.util.List;

public class GetSubListLinkedListExample {

        public static void main(String[] args) {

          //create LinkedList object
          LinkedList lList = new LinkedList();

          //add nodes to LinkedList
          lList.add("1");
          lList.add("2");
          lList.add("3");
          lList.add("4");
          lList.add("5");

          System.out.println("LinkedList contains : " + lList);

          /*
           * To get a sublist from a Java LinkedList, use the
           * List subList(int start, int end) method.
           *
           * This method returns the portion of the Linked List from start index inclusive to end index exclusive.
           */

          List lst = lList.subList(1,4);
          System.out.println("Sublist contains : " + lst);

          /*
           * Please note that sublist is the original list, so any changes made to sublist will also be reflected back to original LinkedList
           */

           //Remove a node from the sublist
           lst.remove(2);
           System.out.println("Sublist now contains : " + lst);
           System.out.println("Original LinkedList now contains : " + lList);
        }
      }
```

```

         /*Output would be:  LinkedList contains : [1, 2, 3, 4, 5]
         Sublist contains : [2, 3, 4]
         Sublist now contains : [2, 3]
         Original LinkedList now contains : [1, 2, 3, 5] */
```

Still more Linked List methods that are contained in the Java API's LinkedList class.

**`size`**`()`
Returns the number of elements in this list.

**`set`**`(int index, E element)`
Replaces the element at the specified position in this list with the specified element.

**`remove`**`()`
Retrieves and removes the head (first element) of this list.

**`removeFirst`**`()`
Removes and returns the first element from this list.

**`removeLast`**`()`
Removes and returns the last element from this list.

**`addFirst`**`(E o)`
Inserts the given element at the beginning of this list.

**`addLast`**`(E o)`
Appends the given element to the end of this list.

**`getFirst`**`()`
Returns the first element in this list.

**`getLast`**`()`
Returns the last element in this list.

**`contains`**`(Object o)`
Returns `true` if this list contains the specified element.

**`clear`**`()`
Removes all of the elements from this list.

`Exceptions thrown…`
`NoSuchElementException` - if this list is empty.
`NullPointerException` - if the specified collection is null.
`IndexOutOfBoundsException` - if the specified index is out of range