Linked Lists

A **linked list** is a low-level data structure. It stores an *ordered* list of items in individual "node" objects that have pointers to other nodes.

In a **singly linked list**, the nodes each have one pointer to the next node.

```csharp
public class LinkedListNode
{
    public int Value { get; set; }

    public LinkedListNode Next { get; set; }

    public LinkedListNode(int value)
    {
        Value = value;
    }
}
```
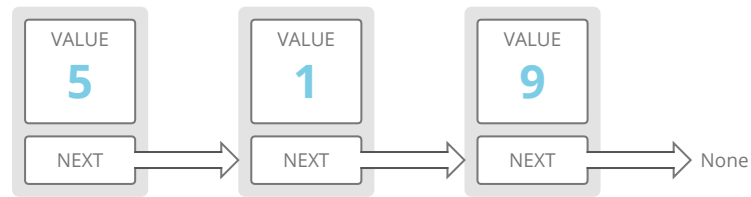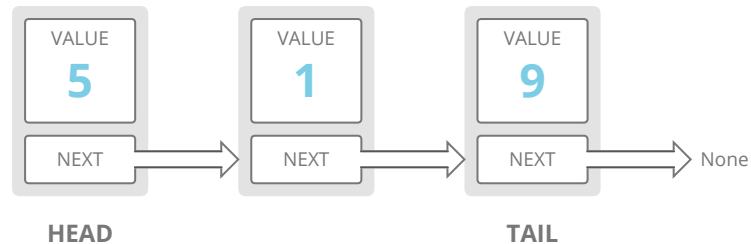C# (beta) ▾

So we could build a singly linked list like this:

```csharp
var a = new LinkedListNode(5);
var b = new LinkedListNode(1);
var c = new LinkedListNode(9);


a.Next = b;
b.Next = c;
```
C# (beta) ▾

In a linked list, the first node is called the **head** and the last node is called the **tail**.

Often, our only connection to *the list itself* is a variable pointing to the head. From there we can walk down the list to all the other nodes.

Like linked lists, **arrays** also store ordered lists of items, so you usually have a choice of which one to use.

Advantages of linked lists:

1. Linked lists have **constant-time insertions and deletions** in any position (you just change some pointers). Arrays require $O(n)$ time to do the same thing, because you'd have to "shift" all the subsequent items over 1 index.
2. Linked lists can **continue to expand** as long as there is space on the machine. Arrays (in low-level languages) must have their size specified ahead of time. Even in languages with "dynamic arrays" that automatically resize themselves when they run out of space (like Python, Ruby and JavaScript), the operation to resize a dynamic array has a large cost which can make a single insertion unexpectedly expensive.

Disadvantages of linked lists:

1. To access or edit an item in a linked list, you have to **take $O(i)$ time to walk from the head of the list to the ith item** (unless of course you already have a pointer directly to that item). Arrays have *constant-time* lookups and edits to the ith item.

Another type of linked list is a **doubly linked list**, which has pointers to the next *and the previous* nodes.

```
public class LinkedListNode
{
    public int Value { get; set; }

    public LinkedListNode Next { get; set; }

    public LinkedListNode Previous { get; set; }

    public LinkedListNode(int value)
    {
        Value = value;
    }
}
```

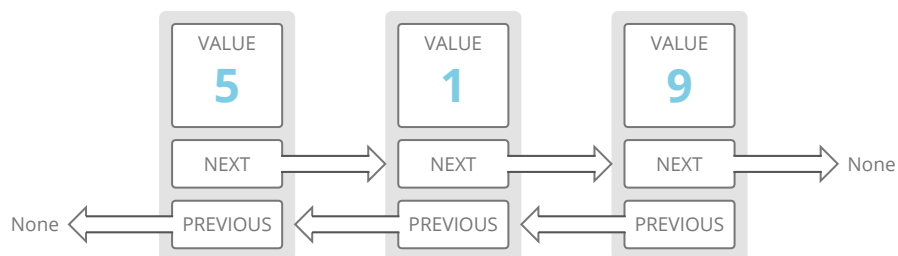So we could build a doubly linked list like this:

```
var a = new LinkedListNode(5);
var b = new LinkedListNode(1);
var c = new LinkedListNode(9);

// Put b after a
a.Next = b;
b.Previous = a;

// Put c after b
b.Next = c;
c.Previous = b;
```

VALUE	VALUE	VALUE
5	1	9
NEXT →	NEXT →	NEXT → None
None ← PREVIOUS	← PREVIOUS	← PREVIOUS

Doubly linked lists allow us to traverse our list *backwards*. In a *singly* linked list, if you just had a pointer to a node in the *middle* of a list, there would be *no way* to know what its previous node was. Not a problem in a doubly linked list.