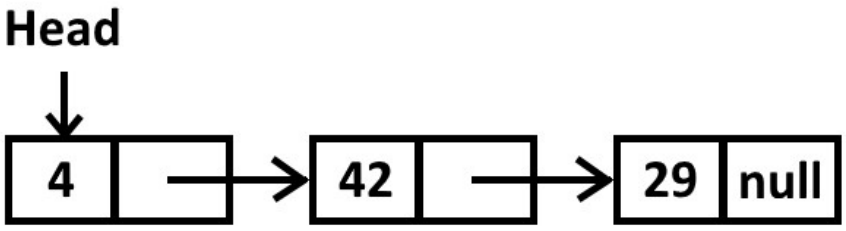
Linked Lists Reading

Tim "Dr. T" Chamillard

What's a Linked List?

A linked list consists of a set of nodes connected by links, like the following picture.



One of the advantages of linked lists over arrays is that the nodes in the list can be scattered around in memory. With an array, all the elements of the array need to be contiguous (next to each other) because of the way indexing into the array works. That means that for large arrays, we need to find a large, contiguous block of memory to allocate for the array. If the memory is fragmented (much like a disk being fragmented) that can be hard to do. We don't have that problem with linked lists.

Of course, that benefit comes at a cost. Linked lists aren't "random access" data structures like an array or a `List`, where we can access any element of the structure in O(1) time. Instead, accessing a particular element in the list will cost us O(n) time because we have to start at the head (or the tail) of the list and traverse the list linearly to get to the element we want to access.

The picture shown above is actually called a singly linked list, where each node in the list only has a reference to the next item in the list. We can also implement doubly linked lists, where each node points to the next item in the list and the previous item in the list. This reading will focus on singly linked lists, but if you understand the ideas here you should be able to extend them to doubly linked lists (or even circular linked lists).

The `System.Collections.Generic` namespace has a `LinkedList` class that you should definitely use in practice when you need a linked list; we're walking through a simpler implementation here to better our understanding of how linked lists work and to do our algorithm analysis on simpler code.

Our Building Blocks

We'll actually use two classes as the basis for our linked list: a `LinkedListNode` and a `LinkedList`. We'll then add child classes for an `UnsortedLinkedList` and a `SortedLinkedList`.

Let's start with the `LinkedListNode` class:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace LinkedLists
{
    /// <summary>
    /// A linked list node
    /// </summary>
    class LinkedListNode<T>
    {
```

We're making our node a generic so it can contain any data type for the node value.

```
        #region Fields

        T value;
        LinkedListNode<T> next;

        #endregion
```

Each node contains a value and a reference to the next node in the linked list.

```
        #region Constructors

        /// <summary>
        /// Creates new node with given value and next node
        /// </summary>
        /// <param name="value">value</param>
        /// <param name="next">next node</param>
        public LinkedListNode(T value, LinkedListNode<T> next)
        {
            this.value = value;
            this.next = next;
        }

        #endregion
```

The constructor is pretty straightforward, initializing the fields to the parameters.

```
        #region Properties

        /// <summary>
        /// Gets the node value
        /// </summary>
        /// <value>node value</value>
        public T Value
        {
            get { return value; }
        }
```

The `Value` property is read-only, so we can get the node value but we can't change it after the node has been created.

```
        /// <summary>
        /// Gets and sets the next node
        /// </summary>
        /// <value>next node</value>
        public LinkedListNode<T> Next
        {
            get { return next; }
            set { next = value; }
        }

        #endregion
    }
}
```

The `Next` property provides both read and write access. We need to read the property when we traverse the list so we can move to the next node in the list. We may need to write the property when we're adding a node to or removing a node from the list.

Okay, on to the `LinkedList` class:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace LinkedLists
{
    /// <remarks>
    /// A linked list of a data type
    /// </remarks>
    abstract class LinkedList<T>
    {
        protected LinkedListNode<T> head;
        protected int count;
```

Our `LinkedList` class is also a generic, of course. The `head` field is standard for linked lists because it refers to the start of the linked list. The `count` field keeps track of how many items (nodes) are in the linked list.

```
       #region Constructor

        /// <summary>
        /// Constructor
        /// </summary>
        protected LinkedList()
        {
            head = null;
            count = 0;
        }
```

```
        #endregion
```

We set the `head` and reference to `null` because there aren't any nodes in the linked list yet. We set `count` to 0 for the same reason.

```
        #region Properties

        /// <summary>
        /// Gets the number of nodes in the list
        /// </summary>
        public int Count
        {
            get { return count; }
        }

        /// <summary>
        /// Gets the head of the list
        /// </summary>
        /// <value>head of the list</value>
        public LinkedListNode<T> Head
        {
            get { return head; }
        }

        #endregion
```

We provide a `Count` property to get the number of nodes in the list and a `Head` property so consumers of the class can traverse the list if they want to.

```
        #region Public methods

        public abstract void Add(T item);
```

The `Add` method is abstract because it will work differently for unsorted and sorted linked lists.

```
        /// <summary>
        /// Removes all the items from the linked list
        /// </summary>
        public void Clear()
        {
            // unlink all nodes so they can be garbage collected
            if (head != null)
            {
```

As the comment says, we want to unlink all the nodes so the garbage collector can recognize that nothing refers to those nodes so they're eligible for collection. We only have to do this if there's at least one node in the list.

```
                LinkedListNode<T> previousNode = head;
                LinkedListNode<T> currentNode = head.Next;
```

Because we're implementing a singly linked list rather than a doubly linked list, the only way to know what the preceding node is for the current node is to save it as we move from the previous node to the current node. That's why we have both `previousNode` and `currentNode` variables above. Note that `currentNode` is `null` at this point if the list only has one node.

```
            previousNode.Next = null;
```

The line above removes the reference from the `previousNode` to the `currentNode`, unlinking the two nodes.

```
            while (currentNode != null)
            {
                previousNode = currentNode;
                currentNode = currentNode.Next;
                previousNode.Next = null;
            }
        }
```

The loop above "walks the list", moving along one node each time and unlinking the two nodes. The loop ends when we've reached the end of the list.

```
        // reset head and count
        head = null;
        count = 0;
    }
```

Finally, we reset head and count to reflect an empty list.

```
    /// <summary>
    /// Removes the given item from the list
    /// </summary>
    /// <param name="item">item to remove</param>
    public bool Remove(T item)
    {
        // can't remove from an empty list
        if (head == null)
        {
            return false;
        }
```

Our first special case when removing a node from the list is that we're trying to remove from an empty list. That's impossible, so we retrun false.

```
        else if (head.Value.Equals(item))
        {
            // remove from head of list
            head = head.Next;
            count--;

            return true;
```

Our second special case that we're removing the head of the list. If we are, we need to change our `head` field to point at the next node in the list because that node is the new head. We also reduce th number of nodes in the list.

```
        }
        else
        {
            LinkedListNode<T> previousNode = head;
            LinkedListNode<T> currentNode = head.Next;
            while (currentNode != null &&
                !currentNode.Value.Equals(item))
            {
                previousNode = currentNode;
                currentNode = currentNode.Next;
            }
```

The code above is very similar to the code in the `Clear` method. The only difference is that the while loop terminates when we reach the end of the list or we've found the node we were looking for.

```
            // check for didn't find item
            if (currentNode == null)
            {
                return false;
            }
```

The `currentNode` variable is only `null` at this point if we reached the end of the list without finding the node we were looking for, so we return `false`.

```
            else
            {
                // set link and reduce count
                previousNode.Next = currentNode.Next;
                count--;

                return true;
            }
        }
    }
```

Finally, we set the previous node's `Next` property to point to the current node's `Next` property, which unlinks the current node from the list. We also reduce the number of nodes in the list.

Because in the worst case we traverse the entire list looking for the node to remove, Remove is an O(n) operation.

```
    /// <summary>
    /// Finds the given item in the list. Returns null
    /// if the item wasn't found in the list
    /// </summary>
    /// <param name="item">item to find</param>
    public  LinkedListNode<T> Find(T item)
```

```
    {
```

The `Find` method is like the `IndexOf` method for our dynamic arrays, but because linked lists don't have indexes for the nodes, we return a reference to the node with the given value. This can be helpful if the consumer of the class wants to traverse the list starting at that node.

You might be wondering why we've included the `Find` method here rather than in the `UnsortedLinkedList` and `SortedLinkedList` child classes (which we'll get to soon) instead. For our dynamic arrays, we could use binary search on our sorted dynamic array to find the item in O(log n) time. Unfortunately, that only works because our dynamic array was a random-access structure. In a linked list, finding a node works the same whether or not the list is sorted.

```
        LinkedListNode<T> currentNode = head;
        while (currentNode != null &&
            !currentNode.Value.Equals(item))
        {
            currentNode = currentNode.Next;
        }
```

The code above walks the list until it reaches the end of the list or finds the item.

```
        // return node for item if found
        if (currentNode != null)
        {
            return currentNode;
        }
        else
        {
            return null;
        }
    }
```

If we found the node, we return it; otherwise, we return `null` to indicate that the node wasn't in the list.

Because in the worst case we traverse the entire list looking for the node, Find is an O(n) operation.

The Unsorted Linked List

Now that we have our generic abstract base class, we can extend it with a concrete class for unsorted linked lists. The only abstract method we need to implement in this class is the `Add` method.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
```

```
using System.Threading.Tasks;

namespace LinkedLists
{
    /// <summary>
    /// An unsorted linked list
    /// </summary>
    class UnsortedLinkedList<T> : LinkedList<T>
    {
        #region Constructors

        public UnsortedLinkedList() : base()
        {
        }

        #endregion
```

As usual, we include a constructor.

```
        #region Public methods

        /// <summary>
        /// Adds the given item to the list
        /// </summary>
        /// <param name="item">item to add</param>
        public override void Add(T item)
        {
```

The only abstract method we need to implement in this class is the `Add` method.

```
            // adding to empty list
            if (head == null)
            {
                head = new LinkedListNode<T>(item, null);
            }
```

Linked lists are full of special cases! If we're creating the first node in the list, we need to make sure the `head` refers to the new node.

```
            else
            {
                // add to front of list
                head = new LinkedListNode<T>(item, head);
            }
```

Because the list is unsorted, we can simply add the new node at the front of the list. We do that by making `head` to refer to a new node that contains the item to be added and refers to the old head of the list.

```
            count++;
        }

        #endregion
    }
```

```
}
```

Because we just added a node to the list, we increment `count`.

All the operations in this method are constant-time operations, so Add is an O(1) operation.

The Sorted Linked List

Here's the code for a sorted linked list:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace LinkedLists
{
    /// <summary>
    /// A sorted linked list
    /// </summary>
    class SortedLinkedList<T> : LinkedList<T> where T:IComparable
    {
        #region Constructors

        public SortedLinkedList() : base()
        {
        }

        #endregion
```

As usual, we include a constructor.

```
        #region Public methods

        /// <summary>
        /// Adds the given item to the list
        /// </summary>
        /// <param name="item">item to add</param>
        public override void Add(T item)
        {
```

The only abstract method we need to implement in this class is the `Add` method.

```
            // adding to empty list
            if (head == null)
            {
                head = new LinkedListNode<T>(item, null);
            }
```

If we're creating the first node in the list, we need to make sure the `head` refers to the new node.

```
            else if (head.Value.CompareTo(item) >= 0)
            {
                // adding before head
                head = new LinkedListNode<T>(item, head);
            }
```

Another special case. Adding to the front of the list works just like it did in the `Add` method for the `UnsortedLinkedList`.

```
            else
            {
                // find place to add new node
                LinkedListNode<T> previousNode = null;
                LinkedListNode<T> currentNode = head;
                while (currentNode != null &&
                    currentNode.Value.CompareTo(item) < 0)
                {
                    previousNode = currentNode;
                    currentNode = currentNode.Next;
                }
```

The code above walks the list looking for the correct place to add the new node.

```
                // link in new node between previous node and current node
                previousNode.Next = new LinkedListNode<T>(item, currentNode);
```

Now we can finally insert the new node into the list between the previous node and the current node. We do that by making the previous node refer to a new node that contains the item to be inserted and a reference to the current node. This works even if `currentNode` is null at this point (meaning the new node should go at the end of the list).

```
            }
            count++;
        }

        #endregion
    }
}
```

Because we just added a node to the list, we increment `count`.

In the worst case, the new node goes at the end of the list, so we need to traverse the entire list to the end. That makes Add an O(n) operation.