Lecture 30 — The Linked List

J. Zarnett
jzarnett@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

March 14, 2015

Acknowledgments: W.D. Bishop

Arrays and Collections

We’ve established that we would like to have a dynamic collection that does not rely on an array.

When using an array, we have a reference to the array.

The reference tells us where the start of the array is in memory.

The index tells us which entry of the array to access.

In a collection, the array serves the purpose of keeping the elements next to one another so that we can move from one to the next.

To find the next object in an array, we increment the index by 1.

Linking Collection Elements

If the elements are not stored in an array, given that we know where the first object is, how might we get to the next object?

Idea: use a reference to link the objects together.

In the simplest case, each object has a reference to the next object.
A null reference indicates there is no next object.

Analogy: pirate treasure map.
Follow the map's directions; it leads to treasure and another map.
The 2^{nd} map says how to find another treasure and a third map.
… and repeat until we get to the final treasure (that has no map).

The Linked List

Linked lists are similar in function to arrays but are more flexible:

- The size of a linked list is not fixed
- Objects may be easily inserted or deleted from a linked list
- Objects may be sorted (if desired)

Linked lists are formed using self-referential classes:
A class that has a reference to an object of the same type.

For example, a class `ListEntry` can contain one or more data fields as well as a reference to another object of type `ListEntry`.

Self-Referential Classes

Wait a minute, we previously said this is a compile-time error:

```
struct Invoice
{
    public int number;
    public Date date;
    public bool paid;
    public Invoice previousInvoice;
};
```

The difference is `struct` is a value type and `class` is a reference type.

A reference variable may be null; a structure variable may not.

Linking Objects

Let's write a very simple `ListEntry` class.

```
public class ListEntry
{
    public int data;
    public ListEntry next;
}
```

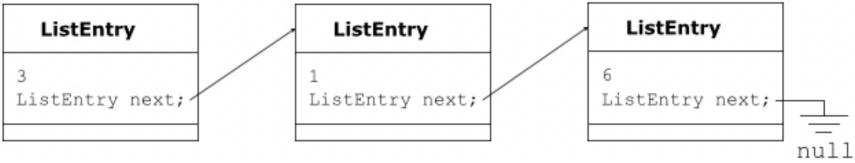
This works because `next` can be null.

The compiler can figure out how much memory `ListEntry` needs because it knows the size of an `int` and the size of a reference (`next`).

Creating a List of Integers

Using the class previously defined, it is possible to create a collection of records known as a singly linked list.

For example, it is possible to define a list containing a set of integers {3,1,6} as follows:

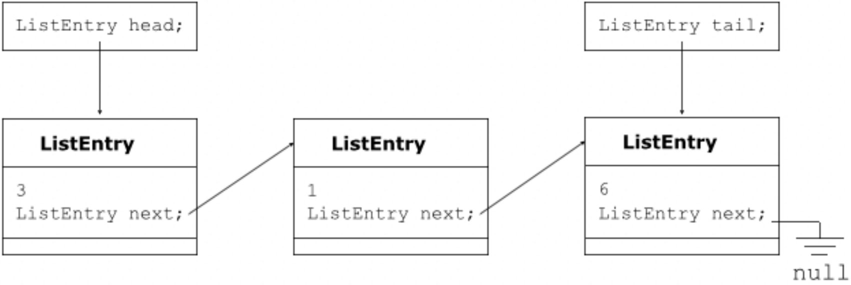


When a reference is null, it is sometimes drawn as if it's an electrical connection to ground.

Accessing the Linked List

The linked list on the previous slide has no references pointing to it...

Developers often keep a reference to the head and the tail of the list:



A tail reference is not strictly needed; it's a perfectly functional linked list with only a head reference. A tail reference can be convenient.

Less Minimal ListEntry

```
class ListEntry
{
    int data;
    ListEntry next;
    public ListEntry( int d )
    {
        data = d;
        next = null;
    }
    public int Data
    {
        get{ return data; }
        set{ data = value; }
    }
    public ListEntry Next
    {
        get{ return next; }
        set{ next = value; }
    }
    public override string ToString( )
    {
        return( data.ToString( ) );
    }
}
```

The `List` Class

Next, a class to store a reference to the head and tail of the linked list.

For the purpose of this example, this class is named `List`.

This class often provides the following features:

- Member fields to refer to the head and the tail of the list
- A constructor
- A `ToString( )` method
- A `Clear( )` method
- An `Append( )` method
- A `DeleteFirst( )` method
- A `Prepend( )` method
- Properties for accessing the private fields

The List Class

The List class is too large to fit on a single slide, but let’s show the different parts.

```
class List
{
  private class ListEntry
  {
      // It is legal to have ListEntry nested inside List
      // In fact, if ListEntry makes no sense outside of
      // a List, this is good practice.
      // Not shown for space reasons
  }

  // Private member variables
    private ListEntry head;
    private ListEntry tail;

    // Constructor
    public List( )
    {
        head = null;
        tail = null;
    }
```

The List Class : Properties

```
// Properties
public int Head
{
    get{ return head.Data; }
}

public int Tail
{
    get{ return tail.Data; }
}

public bool IsEmpty
{
    get{ return( head == null ); }
}
```

The List Class : ToString

```
// Traverse the list starting at the head
// Stop traversing when a null reference is found
public override string ToString( )
{
    string tmp = "Head -> ";

    ListEntry current = head;
    if( current == null )
    {
        tmp = "null";
    }
    while( current != null )
    {
        tmp += current + " -> ";
        current = current.Next;
    }
    return( tmp );
}
```

The List Class : Clear

```
// The implementation here is trivial.
// By setting head and tail to null, any list entries
// will be garbage and subject to garbage collection.
public void Clear( )
{
    head = null;
    tail = null;
}
```

The List Class : Append

```
// Add entry to the end of the list
// Special case: list is empty
public void Append( int i )
{
    ListEntry tmp = new ListEntry( i );

    // Done by default, but here for clarity
    tmp.Next = null;

    if( head == null )
    {
        head = tmp;
    }
    else
    {
        tail.Next = tmp;
    }
    tail = tmp;
}
```

The `List` Class : Prepend

```
// Add entry to the start of the list
// Special case: list is empty
public void Prepend( int i )
{
    ListEntry tmp = new ListEntry( i );

    tmp.Next = head;

    if( head == null )
    {
        tail = tmp;
    }
    head = tmp;
}
```

The List Class : Delete First

```
   // Delete the first entry where the data is equal to i
    public void DeleteFirst( int i )
    {
        ListEntry current = head;
        ListEntry previous = null;

        while( current != null && current.Data != i )
        {
            previous = current;
            current = current.Next;
        }
        if( current == null )
        {
            throw new ArgumentException( "List entry not found" );
        }
        if( current == head ) // Special case: deleting first entry
        {
            head = current.Next;
        }
        else
        {
            previous.Next = current.Next;
        }
        if( current == tail ) // Special case: deleting last entry
        {
            tail = previous;
        }
    }
} // End of List class
```

List Creation Demonstration

Let’s work with this code and represent visually what the following series of statements would look like when executed.

```
List list = new List();
list.Append( 3 );
list.Append( 1 );
list.Append( 6 );
list.Prepend( 4 );
list.Prepend( 5 );
list.DeleteFirst( 4 );
list.Prepend( 2 );
list.Clear( );
```

[Demo done on the board to facilitate understanding.]

Other Methods of `List`

It may be desirable to add methods for searching, sorting and inserting new items:

- `void Sort( )`
- `ListEntry FindFirst( )`
- `ListEntry FindNext( )`
- `void InsertBefore( )`
- `void InsertAfter( )`
- `bool Contains( )`

It should be noted that many of these methods are similar in complexity to the `DeleteFirst( )` method.

Also, it should be noted that more error checking could be added to the methods that have been presented.

Expanding on the Simple Linked List

A second reference in each `ListEntry` that refers to the previous element can simplify deletion and other complex operations.

This is known as a doubly-linked list.

The use of dummy items (i.e., dummy head or dummy tail) can simplify some the termination conditions for some loops.

A dummy item is a `ListEntry` that has a valid reference but never any valid data. It is constructed when the list is constructed.

A length member field can be added to keep track of the length of the list as items are added and removed.