7SENG010W Data Structres & Algorithms

Week 3 Lecture

Linked Lists

Overview of Week 3 Lecture: Linked Lists

Aim is to introduce *list* data structures & the main algorithms that are applied to them[1]:

- *List Data Structures*
  - Properties of Lists
  - Singly linked lists
  - Doubly linked lists
- *List Operations*
  - Creation, insertion & deletion
- *.NET List classes*
  - `List<T>` class
  - `LinkedList<T>`, `LinkedListNode<T>` classes

[1] Acknowledgements: these notes are partially based on those of P. Brennan & K. Draeger.

PART I

List Data Structures

Properties of List Data Structures

Lists are relatively simple *collection* data structures:

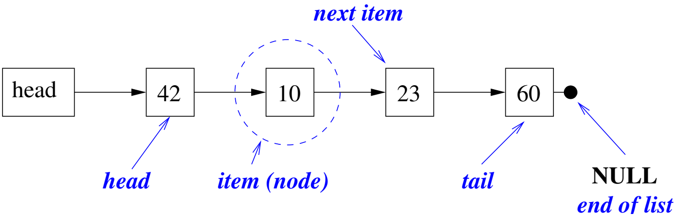
- *Linear* data structures: organised as a *sequence* of data items.
- *Dynamic* data structures:
  - a sequence of data items that *does not have a fixed length*,
  - new data items can be *added* & existing data items can be *deleted* from it.
- *Linked list* data structures: a sequence of "*data nodes*" connected by "*links*" to *adjacent nodes* in the list.
  - Data nodes are *accessed* by following the links between the nodes.
  - Data items (nodes) in a list are often *identified* or *found* by being in a "*significant*" or "*relative*" position within a list.
  - *Significant* list positions are the *head* (front or first) of a list, the *tail* (last or end) of a list.
  - *Relative* list positions are relative to the *current node*: the *previous node* & the *next node*.
- *Representations*: two types of lists are used: *Singly linked* lists & *Doubly linked* lists.

Singly Linked Lists

Lists are one of the simplest "*collection*" type data structures used in programming, and it has a simple definition:

Definition: List
A collection of items accessible one after another beginning at the **head** and ending at the **tail**.

The **head** is the first item in the list. The **tail** is the **last** item in the list[2].

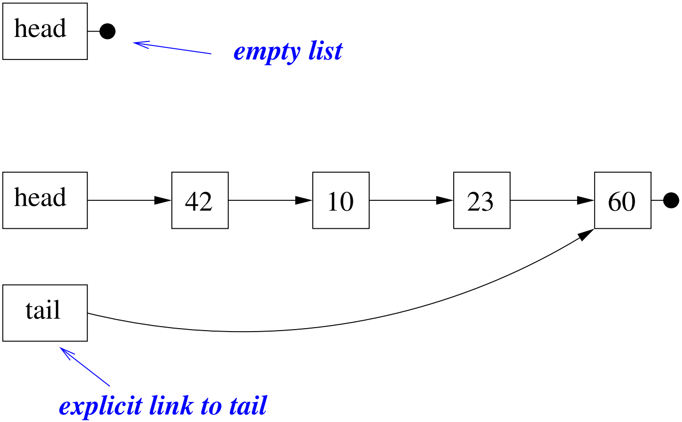


This is an example of a (*singly linked*) list containing the numbers: 42, 10, 23, 60, where 42 is the **head** and 60 is the **tail** of the list.

[2] **Alternative definition of "tail":** all but the first item of a list; the list following the head.

Empty List & List with Tail

Below are examples of an **empty list** and a list with an additional **tail** link, rather than simply relying on having to traverse the list to get to the last item.



Definition of a List

A **linked list** or just **list** consists of a collection of any number of data items of the same data type in which items may be:

- *inserted*
- *deleted*

at **any point in the list**.

Since *insertion* and *deletion* can occur **anywhere** in a list, this data structure is very general with only a few restrictions on its **structure** or **operations**.

For example, insertion and deletion can occur at **both ends of a list**, as well as **anywhere in between**.

Unlike some of the data structures we shall see, e.g. trees, which have a lot of restrictions on both their structure and operations.

Additional List Operations

Other common operations that can be performed on a list are:

- **clear** – remove all of the items in the list
- **first, last** – return the value of the first/last item in the list, **but do not remove it from the list**.
- **length** – return the number of items in the list, an empty list is length 0, previous list has length 4.
- **search** – search for an item in the list, by traversing the list, usually working from the head to the tail.
- **isEmpty** – indicate if the list is empty, i.e. no items in the list.

For comparison see the list methods of the `LinkedList<T>` class below, or Java's [`List class`].

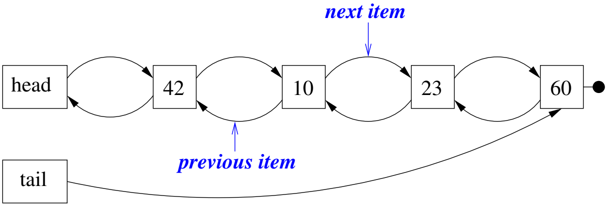
Doubly Linked Lists

The lists so far are known as **singly linked** lists, as each node only has **one link** (reference/pointer) to the **next** (or successor) node in the list.

But an application may require fast access to the **previous** (or predecessor) node in the list.

Solved by adding to each node a **second link** (reference/pointer) to its predecessor node, hence a **doubly linked** list, or **two-way** list.

Below is the previous list as a **doubly linked** list.

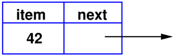


List Node Representations

- *Singly linked list nodes* have the following structure after creation & after insertion into a list:

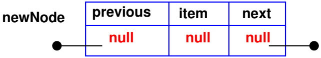
ListNode newNode = new ListNode();

ListNode newNode = new ListNode();
newNode .item = 42 ; etc

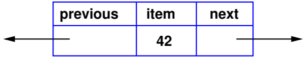


- *Doubly linked list nodes* have the following structure after creation & after insertion into a list:

ListNode newNode = new ListNode();



ListNode newNode = new ListNode();
newNode .item = 42 ; etc



Note: for simplicity node's `previous` & `next` fields are not included in diagrams.

Singly versus Doubly Linked Lists

A further advantage of using a *doubly linked* list is that the algorithms for *inserting* & *deleting* nodes are simplified in that there is no need for a "*previous*" node link to be managed when stepping through the list.

In contrast a *doubly linked list* has the disadvantage that it carries the *overhead of additional memory space* used to store the extra link (pointer/reference) to the previous node.

However, this disadvantage is usually seen as being outweighed by the above advantages.

List Algorithms

Linked List algorithms usually have the following components:

- `while`-loops or `for`-loops:
  - Each iteration "processes" the *current* nodes data, then moves to the *next* node in the list.
  - Terminating when either a specific node is found, e.g. a search, or when the *end of the list* is reached, i.e. `next == null`.
- The *meta data* is usually *references* (pointers) to the: *current* node, *previous* node & *next* node.
- The order of complexity Big-O for (non-sorting) operations on a list of *N* items/nodes is either *Constant – O(1)*, e.g. insert an item at the head of the list, or *Linear – O(N)*, e.g. delete an item from the list.

The operations we shall focus on are:

- *inserting* an item into a list,
- *deleting* an item from a list,
- *searching* for a particular item/node is required for *insertion after* an item & *deletion* of an item.

PART II

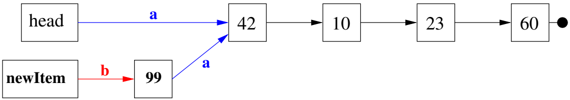
Singly Linked List Operations

Insertion & Deletion

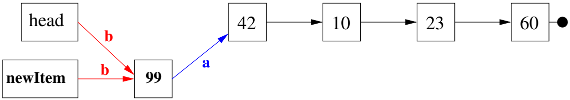
Singly Linked List Operation: Insert Item at Head

InsertAtHead(99) insert 99 into example list as the new head of the list.

- *Step 1:* create a `newItem` node for 99 (link b), setting `newItem.next` to `head`, i.e. link a for current head node 42:



- *Step 2:* insert `newItem` 99 node into the list by setting `head` to `newItem`, i.e. replace link a by link b:

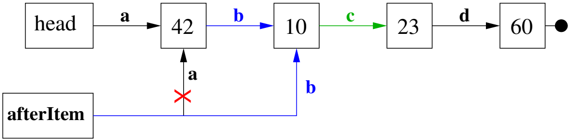


- *Step 3:* Completed insertion of 99 as new head of list, with resultant list:

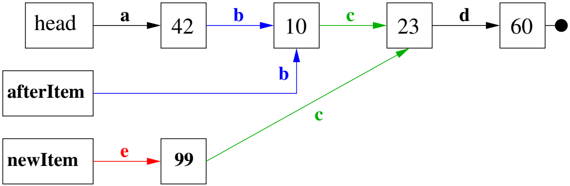
Singly Linked List Operation: Insert After Item (1/2)

Perform *InsertAfter(99, 10)* – insert 99 into example list after 10.

- *Step 1:* search for the `afterItem` node containing 10, return its link b:

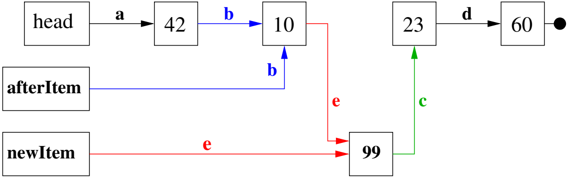


- *Step 2:* create a `newItem` node for 99 (link e), setting `newItem.next` to `afterItem.next`, i.e. link c:

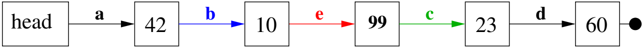


Singly Linked List Operation: Insert After Item (2/2)

- *Step 3:* insert `newItem` node containing 99 into the list by setting `afterItem.next` to `newItem`, i.e. overwrite link c with link e:



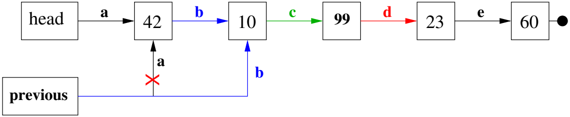
- *Step 4:* Completed insertion of 99 after 10, with resultant list:



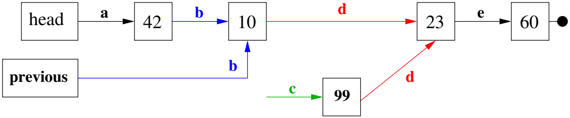
Singly Linked List Operation: Delete Item

Perform *Delete(99)* – delete 99 from the list.

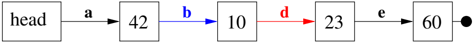
- *Step 1:* search for the `previous` node to 99 which is 10, since `previous.next.item == 99` & return its link b:



- *Step 2:* disconnect node 99 from list by setting `previous.next` (99) to `previous.next.next` (23), i.e. replace 10's link c with link d:



- *Step 3:* Completed deletion of 99, with resultant list:



Singly Linked List: Implementation

When implementing a *singly linked list* two classes need to be defined:

- Firstly, we must defined the list's *node* data structure, this is the `ListNode` class, it has to contain:
  - *Data* that is to be stored in the list's node.
    We shall use `Object` as its type.
  - *Link* to the next node in the list if it exists or *null* if it does not.
    The link's type must be a *reference* to another list *node*, i.e. of type `ListNode`.
- Secondly, the *linked list* class itself `LinkedList`, this has to contain:
  - The *list*, all of the list's nodes in a sequence.
    This is achieved by having a link to the *head* (first) node in the list if it exists or *null* if the list is empty.
    This is of type `ListNode`.
  - Optionally, it may also contain:
    A *link* to the *tail* (last) node in the list if it exists or *null* if the list is empty.
    A count of the number of items (nodes) in the list.
  - *Methods* required for: creation, node insertion & deletion, searching, etc.

Singly Linked List: `ListNode` class

```
class ListNode
{
  private Object   item ;    // node's "data"
  private ListNode next ;    // node's "link" to next node in list

  public ListNode(){
    item = null ;
    next = null ;
  }

  public ListNode( Object item ){
    this.item = item ;
    this.next = null ;
  }

  public ListNode( Object item, ListNode next ){
    this.item = item ;
    this.next = next ;
  }

  public void setItem( Object item ){  this.item = item ;  }

  public void setNext( ListNode next ){  this.next = next ;  }

  public Object getItem(){  return this.item ;  }

  public ListNode getNext(){  return this.next ;  }

} // ListNode
```

Singly Linked List: `LinkedList` class (1/4)

```
class LinkedList
{
  protected ListNode head   = null ;   // points to the head of the list
  protected int      length = 0    ;   // number of nodes in the list

  public LinkedList()
  {
    head   = null ;                    // empty list
    length = 0    ;                    // no nodes in the list
  }

  public bool isEmpty()
  {
    return ( length == 0 ) ;           // or ( head == null )
  }

  public void insertAtHead( Object item )
  {
    ListNode newItem = new ListNode( item, head ) ;

    head = newItem ;
    length++ ;
  }
```

Singly Linked List: `LinkedList` class (2/4)

This `insertAfter(newItem, afterItem)` operation uses a private helper method `findItem(afterItem)` to find the node that the `newItem` is to be inserted after.

```
public bool insertAfter( Object newItem, Object afterItem )
{
   // find the afterItem's node
   ListNode afterNode = findItem( afterItem ) ;

   if ( afterNode != null )
   {  // afterItem is in list

      // create newItem's node & set its next to afterItem's next

      ListNode newItemNode = new ListNode( newItem, afterNode.getNext() ) ;

      // insert newItem's node into the list after afterItem's node
      afterNode.setNext( newItemNode ) ;
      length++ ;

      return true ;
   }
   else
   {    // afterItem not in list, insertion failed
        return false ;
   }
}
```

Singly Linked List: `LinkedList` class (3/4)

Use this method to find the node, e.g. `afterItem`, that a new item is to be inserted after.

```
private ListNode findItem( Object item )
{
  // check if list is empty
  if ( !isEmpty() )
  {
     // traverse the list by starting at the head
     ListNode current = new ListNode() ;
     current = head ;

     // while not at end of the list & not found item continue
     while ( (current != null) && ( !(item.Equals( current.getItem()))) )
     {
       current = current.getNext() ;
     }
    return current ;    // the item's node or null if item not found
  }
  else
  {    // list is empty
       return null ;
  }
}
```

Singly Linked List: LinkedList class (4/4)

```
  public void printList()
  {
     if ( header == null )
     {
         Console.WriteLine( "List is empty" ) ;
     }
     else
     {
         ListNode current = new ListNode() ;
         current = head ;

         Console.WriteLine( "Items in the list are:" ) ;

         while ( current != null )    // not at end of the list
         {
            Console.WriteLine( current.item.ToString() ) ;
            current = current.next ;
         }
     }
  }

   // deleteItem( Object item ) & other methods left as an Exercise.

}  //LinkedList
```

PART III

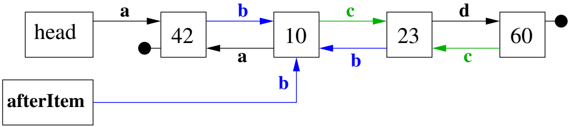
Doubly Linked Lists Operations

Insertion & Deletion

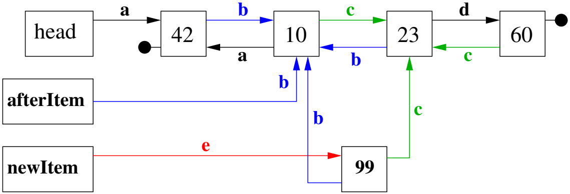
Doubly Linked List Operation: Insert After Item (1/2)

Perform *InsertAfter(99, 10)* – insert 99 into example list after 10.

- *Step 1:* search for the `afterItem` node containing 10, return its link b:

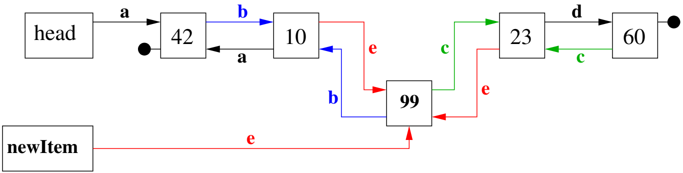


- *Step 2:* create a `newItem` node for 99 (link e), & link it to its *previous* & *next* nodes. Set `newItem.previous` to `afterItem` (link b) & `newItem.next` to `afterItem.next` (link c):

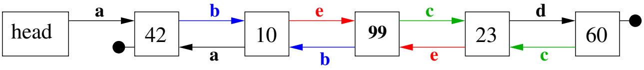


Doubly Linked List Operation: Insert After Item (2/2)

- *Step 3:* connect the *previous* (10) & *next* (23) nodes to the `newItem` (99) node via link e.
  By setting `newItem.previous.next` to `newItem`, i.e. overwrite 10's link c with link e, & `newItem.next.previous` to `newItem`, i.e. overwrite 23's link b with link e.
  The `afterItem` node is no longer needed as link b is available as `newItem.previous`.



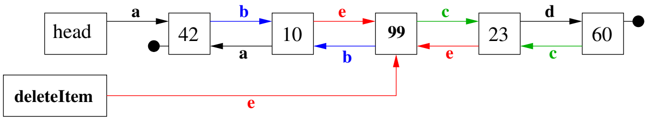
- *Step 4:* Completed insertion of 99 after 10, with resultant list:



Doubly Linked List Operation: Delete Item (1/2)

Perform *Delete(99)* – delete 99 from the list.

- *Step 1:* search for the node to delete, i.e. 99, & return its link e.



Note that for *doubly linked lists* we search for the *node to delete directly* & **not** for its *previous* node, as was the case with *singly linked lists*.

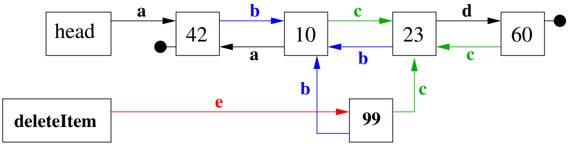
This is because the *previous* node can be access via the delete node's `deleteItem.previous` link, e.g. link b in the above diagram.

Doubly Linked List Operation: Delete Item (2/2)

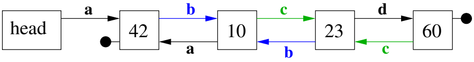
- *Step 2:* disconnect `deleteItem` node 99 from list by unlinking it from its *previous* (10) & *next* (23) nodes.

  Set `deleteItem.previous.next` to `deleteItem.next`,
  i.e. overwrite 10's link e with link c.

  Set `deleteItem.next.previous` to `deleteItem.previous`,
  i.e. overwrite 23's link e with link b:



- *Step 3:* Completed deletion of 99, with resultant list:



Doubly Linked List: DLListNode class

```
class DLListNode
{
  private Object     item ;     // node's "data"
  private DLListNode previous ; // node's link to previous node
  private DLListNode next ;     // node's link to next node

  public DLListNode(){
    item     = null ;
    previous = null ;
    next     = null ;
  }

  public DLListNode( Object item, DLListNode previous, DLListNode next ){
    this.item     = item ;
    this.previous = previous ;
    this.next     = next ;
  }

  public void setItem    ( Object item ){  this.item = item ;  }
  public void setPrevious( DLListNode previous ){  this.previous = item ;  }
  public void setNext    ( DLListNode next ){  this.next = next ;  }

  public Object     getItem(){  return this.item ;  }
  public DLListNode getNext(){  return this.previous ;  }
  public DLListNode getNext(){  return this.next ;  }

  public void print(){ ... }

} // DLListNode
```

Doubly Linked List: DLinkedList class

```
class DLinkedList
{
  // constants NO_NODE, NO_PREVIOUS_NODE, NO_NEXT_NODE = null ;
  protected DLListNode head   = NO_NODE;    // points to head of list
  protected int        length = 0 ;         // number nodes in list

  public DLinkedList(){
        head = NO_NODE ;   length = 0 ;     // empty list, 0 nodes
  }

  public void insertAtHead( Object item )
  {
     DLListNode newItemNode = new DLListNode( item, NO_PREVIOUS_NODE, head ) ;

     if ( head != NO_NODE)                          // check not empty list
     {
        head.setPrevious( newItemNode ) ;     // link current head to new head
     }
     head = newItemNode ;                           // make newItemNode the head node
     length++ ;
  }

  public  bool        isEmpty(){ ... }
  private DLListNode findItem( Object item ){ ... }
  public  bool        insertAfter( Object newItem, Object afterItem ){ ... }
  public  bool        DeleteItem( Object deleteItem ){ ... }
  public  void        printList(){ ... }

} // DLinkedList
```

PART IV

C♯/.NET List Classes

`List<T>`

`LinkedList<T>`

C♯ List Class: `List<T>`

- There is a C♯ *generic list* class `System.Collections.Generic.List<T>`.
- Since `List<T>` is a *generic* class a *type parameter* must be provided for `T`, to indicate the *type of items* in the list, e.g. `string`, `Object`, etc.
- In otherwords, *type parameter* `T` represents the *data type*, e.g. a basic type or a class type, that will be stored in the list.
- So an instance of `List<T>` is a *variable sized list of objects of type* `T`.
- **However**, it is **not** a "*linked list*" in the sense that we have looked at earlier in the lecture, i.e. a list of *data nodes* connected & accessed by using the *links* between them.
- It is just a *generic* version of the `ArrayList` data structure from the previous lecture, i.e. you access the elements of `List<T>` using an *index*, e.g. `myList[0]`, `myList[1]`, etc, **not via links**.
- See the `List<T>` class documentation for details & example programs.

C♯ Linked List Class: `LinkedList<T>`

However, C♯ does have a real *generic doubly linked list* class that is in the [System.Collections.Generic]() namespace.

It is implemented using the following generic node & list classes:

- `LinkedListNode<T>`
  - This is the *type of a node* that must be used in a list of type `LinkedList<T>`.
  - When instantiating a node object, a type parameter `T`, must be supplied & indicates the type of data stored in the node object.
  - This class **cannot be inherited**, i.e. sub-classed[3].
- `LinkedList<T>`
  - This is the type of a *doubly linked list*.
  - The nodes in this list are of type `LinkedListNode<T>`.

[3] See `sealed` class.

Node Class: `LinkedListNode<T>`

`LinkedListNode<T>` is the *node type* for a `LinkedList<T>` list.

Its *Properties* are:

- List – *gets* the `LinkedList<T>` that the `LinkedListNode<T>` belongs to.
- Next – *gets* the next node in the `LinkedList<T>`.
- Previous – *gets* the previous node in the `LinkedList<T>`.
- Value – *gets* & *sets* the value contained in the node.
- ValueRef – *gets* a reference to the value held by the node.

Useful *methods*:

- Equals(Object) – test if the specified object is equal to the current object.
- GetType() – gets the *Type* of the current instance.
- ToString() – returns a string representation of the current object.

List Class: `LinkedList<T>`

`LinkedList<T>`

Is the *list type* for a *doubly linked list* of `LinkedListNode<T>`.

Its *Properties* are:

- Count – *gets* the *number of nodes* in the `LinkedList<T>`.
- First – *gets* the *first* node in the `LinkedList<T>`, i.e. head of list.
- Last – *gets* the *last* node in the `LinkedList<T>`, i.e. tail of list.

Examples of `LinkedList<T>` Class's Methods

Since the `LinkedList<T>` class is a real list class it has methods for *adding* & *removing* nodes from the list; & *searching* for values in the list.

But it **does not have any sorting methods**.

```
// Adding items/nodes to the list
public void AddBefore( LinkedListNode<T> node, LinkedListNode<T> newNode ) ;
public void AddAfter( LinkedListNode<T> node, LinkedListNode<T> newNode ) ;

public LinkedListNode<T> AddFirst( T value ) ;
public LinkedListNode<T> AddLast( T value ) ;

// Removing items/nodes from the list
public bool Remove(T item) ;       // 1st occurrence of item from list
public void RemoveFirst() ;        // head node
public void RemoveLast() ;         // tail node

// Searching methods
public bool Contains(T item) ;                // checks if item in list

public LinkedListNode<T> Find(T item) ;       // find first item in list
public LinkedListNode<T> FindLast(T item) ;   // find last item in list
```

See the [`LinkedList<T>`] class for a full list of methods.

Example of `LinkedListNode<T>`, `LinkedList<T>` Classes

Using an instance of `LinkedListNode<T>` with `T` as `string`, to represent a list of Zoo animals.

```
string[] bigCats = { "Lion", "Tiger" } ;     // Create some animals

// Create a list of big cat Zoo animals
LinkedList<string> Zoo = new LinkedList<string>( bigCats ) ;

Zoo.AddLast( "Gorilla" ) ; // Add "Gorilla" to end of the list

// Create eagle node for Zoo list
LinkedListNode<string> eagle = new LinkedListNode<string>( "Eagle" ) ;

// Add Eagle before last (Gorilla) node
Zoo.AddBefore( Zoo.Last, eagle  ) ;

// print out the animals in the zoo
foreach ( string animal in Zoo )
        Console.WriteLine( animal ) ;

Zoo.RemoveFirst() ;            // Removes Lion from Zoo
Zoo.Remove( "Gorilla" ) ;      // Removes Gorilla from Zoo

Console.WriteLine( "The Zoo has an Eagle: {0}",
                   Zoo.Contains( "Eagle" )       ) ;
```

For more examples see the [`LinkedList<T>`] class.

The Zoo lists

The sequences of `LinkedList<string>` lists & the structure of their underlying *doubly linked lists*, with the **Count**, **First** (head) & **Last** (tail) properties, produced by the example code: