Contents

Doubly Linked List

Here is *yet* another implementation of the list abstract data type, using *doubly linked list*.

```
using System;
using System.Collections;
using System.Collections.Generic;

public class DLList<T> : ICollection<T>
{
  private Cell head;
  private Cell tail;

  public DLList()
  {
    head = null;
    tail = null;
  }

  private class Cell
  {
    public T Data { get; set; }
    public Cell Next { get; set; }
    public Cell Previous { get; set; }

    public Cell(
      T dataP = default(T),
      Cell previousP = null,
      Cell nextP = null
    ) // We provide default values
    {
      Data = dataP;
      Previous = previousP;
      Next = nextP;
    }
  }

  // Empty
  public bool IsEmpty()
  {
```

```
  return head == null;
}

// Add "to the right",
// at the end of the list.
public void Add(T value)
{
  if (IsReadOnly)
  {
    throw new InvalidOperationException(
      "List is read-only."
    );
  }
  if (head == null)
  {
    head = new Cell(value, null, null);
    tail = head;
  }
  else
  {
    tail.Next = new Cell(value, tail, null);
    tail = tail.Next;
  }
}

public void Clear()
{
  head = null;
  tail = null;
}

public bool Contains(T value)
{
  bool found = false;
  Cell cCell = head;
  while (cCell != null && !found)
  {
    if (cCell.Data.Equals(value))
    {
      found = true;
    }
    cCell = cCell.Next;
  }
  return found;
}
```

```
// Copies the elements of the ICollection to an Array,
 ↪  starting at a particular Array index.
public void CopyTo(T[] array, int arrayIndex)
{
  if (array == null)
    throw new ArgumentNullException(
      "The array cannot be null."
    );
  if (arrayIndex < 0)
    throw new ArgumentOutOfRangeException(
      "The starting array index cannot be negative."
    );
  if (Count > array.Length - arrayIndex)
    throw new ArgumentException(
     "The destination array has fewer elements than the
      ↪  collection."
    );

  Cell cCell = head;
  int i = 0; // keeping track of how many elements were
   ↪  copied.
  while (cCell != null)
  {
    array[i + arrayIndex] = cCell.Data;
    i++;
    cCell = cCell.Next;
  }
}

public bool Remove(T value)
{
  if (IsReadOnly)
  {
    throw new InvalidOperationException(
      "List is read-only"
    );
  }
  bool removed = false;
  if (!IsEmpty())
  {
    // If the value we are looking for
    // is held by head
    if (head.Data.Equals(value))
    {
      head = head.Next;
      // If there was more than one
```

```
      // cell in our list
      if (head != null)
      {
        // We delete the reference
        // to the node
        // we want to remove.
        head.Previous = null;
      }
      else
      {
        // Since there was only one cell in our list,
        // the tail needs to be updated.
        tail = null;
      }
      removed = true;
    }
    else
    {
      Cell cCell = head;
      while (cCell.Next != null && !removed)
      {
        if (cCell.Next.Data.Equals(value))
        {
          cCell.Next = cCell.Next.Next;
          // We test if we reached the end of the list
          if (cCell.Next != null)
          {
            cCell.Next.Previous = cCell;
          }
          else
          {
            // If we did, we update the tail.
            tail = cCell;
          }
          removed = true;
        }
        else
        {
          // If we did not find the value,
          // we move the cCell to the next
          // cell to continue searching.
          cCell = cCell.Next;
        }
      }
    }
  }
```

```
  return removed;
}

public int Count
{
  get
  {
    int size = 0;
    Cell cCell = head;
    while (cCell != null)
    {
      cCell = cCell.Next;
      size++;
    }
    return size;
  }
}

public bool IsReadOnly { get; set; }

public IEnumerator<T> GetEnumerator()
{
  Cell cCell = head;
  while (cCell != null)
  {
    yield return cCell.Data;
    cCell = cCell.Next;
  }
}

IEnumerator IEnumerable.GetEnumerator()
{
  return this.GetEnumerator(); // call the generic
   ↪  version of the method
}

/* We are done realizing the ICollection class. */

public override string ToString()
{
  string returned = "———";
  // Line above the table
  for (int i = 0; i < Count; i++)
  {
    returned += "————";
  }
```

```
    returned += "\n| ";
    // Content of the CList
    Cell cCell = head;
    while (cCell != null)
    {
      returned += $"{cCell.Data} | ";
      cCell = cCell.Next;
    }
    returned += "\n———";
    // Line below the table
    for (int i = 0; i < Count; i++)
    {
      returned += "————";
    }

    // We go through the list in the other direction:

    returned += "\n———";
    // Line above the table
    for (int i = 0; i < Count; i++)
    {
      returned += "————";
    }
    returned += "\n| ";
    // Content of the CList
    cCell = tail;
    while (cCell != null)
    {
      returned += $"{cCell.Data} | ";
      cCell = cCell.Previous;
    }
    returned += "\n———";
    // Line below the table
    for (int i = 0; i < Count; i++)
    {
      returned += "————";
    }

    if (Count > 0)
    {
      returned += $"\nHead: {head.Data}\n";
      returned += $"Tail: {tail.Data}\n";
    }
    return returned;
  }
```

```
public void RemoveF()
{
  if (head != null)
  {
    if (head.Next != null)
    {
      head.Next.Previous = null;
    }
    head = head.Next;
  }
}

public void RemoveL()
{
  if (tail != null)
  {
    if (tail.Previous != null)
    {
      tail.Previous.Next = null;
    }
    tail = tail.Previous;
  }
}

// Method to remove the nth element if it exists.
public void RemoveI(int index)
{
  if (index > Count || index < 0)
  {
    throw new IndexOutOfRangeException();
  }
  else // Some IDE will flag this "else" as redundant.
  {
    if (index == 0)
    {
      RemoveF();
    }
    else if (index == (Count - 1))
    {
      RemoveL();
    }
    else
    {
      int counter = 0;
      Cell cCell = head;
      while (counter < index - 1)
```

```
        {
          cCell = cCell.Next;
          counter++;
        }
        cCell.Next = cCell.Next.Next;
        cCell.Next.Previous = cCell;
      }
    }
  }
}
```

(Download this code)

The main differences with *singly* linked list are as follows:

- Instead of keeping only track of the first element (`first`), we keep track of both the first (`head`) and last (`tail`) elements.
- Each `Cell` contains a pointer to the "next" element (as before), but also to the "previous" element.
- Adding (or removing) to the right is now done in constant time, instead of linear time.
- Traversing the list in opposite order (from end to beginning, or right to left) is now straightforward (cf. the `ToString` method above).
- The rest of the edits are about bookkeeping the `Previous` and `Next` attributes of the `Cell`, as well as updating the `tail` attribute. This makes removing and adding "in the middle of the list" a bit tricky.