Chapter 35

Linked Lists

A linked list is an alternative to an array. It's a list of items where each is "linked" to the next with a pointer. Sometimes it's a better way to store things than an array. It's a basic data structure that all programmers learn.

But mostly it's a good exercise. Playing with linked lists is a traditional way to really get good with pointers. I've also got some fun functions, a new type of loop, and using templates (generics.)

35.1 Simple linked list

An array makes a single chunk of memory with each item next to the other. We can find `A[5]` because it's exactly 5 ints past `A[0]`. We can just jump to it using math.

A linked list doesn't bother putting them in order. The items are scattered around, with each one pointing to the next. We call the item plus the pointer a **node** (but you know it's just a small class.) Linked lists aren't built-in the way arrays are. They're just an idea that we have to make ourselves.

A linked list named `L1` holding 2, 7, 4, 9 could look like this. The lines stand for pointers:

```
        9-->null
         \
L1--> 2   4
       \ /
        7
```

I moved them around to emphasize how we need to use the pointers to find which item is next.

Each node has two things – the value and the pointer – so we have to make a class:

```
class IntNode {
  public int val;
  public IntNode next; // pointer to next one, or null for last
}
```

Our picture written out as IntNodes would look like this. Each number is a created IntNode, and our link to it is `IntNode L1;`, which is aimed at the 2:

```
     --------       --------       -------       -------
L1->|val next|  /->|val next|  /->|val next|  /->|val next|  /->null
    | 2    o-|-/   | 7    o-|-/   | 4    o-|-/   | 9    o-|-/
     --------       --------       --------       --------
```

If you look at the second field, `public IntNode next;`, you might think "wait – we have an IntNode inside of another IntNode?" No – the picture is the right way to think of it.

If you remember from the pointer chapter, a quirk of C# (and Java) is there are two ideas of pointer variables. The "normal" way is we own it, it's inside of us, and we have to create it with `new`. The other is a real pointer: we don't create it, we just use it to point to what someone else made.

`next` is a real pointer. Each IntNode "knows about" the one after it, using `next`.

35.1.1 Linked List loops

I'm going to save how we make these for later, since using them is a better way to understand them.

A standard linked list loop starts a pointer on the first item (called the *head*) and follows it one node at a time until we go off the edge with `null`.

This loop prints the list. For example `printList(L1);`:

```
void printList(IntNode head) {
  IntNode p = head; // loop variable. Aim at 1st item
  while(p!=null) { // not past the end
    print(p.val);
    p=p.next; // go to next node
  }
}
```

We can rewrite it as a `for` loop:

```
for(IntNode p=head; p!=null; p=p.next)
  print(p.val);
```

This isn't even abusive – it's a perfect use of a `for`. In the parens we can clearly see `p` is the loop variable, then where it starts, when it stops and how it

moves. Down below we can focus on only what we do as `p` hits each node.

Here's a quick review of pointers and how they work in that loop. Nothing new:

- Remember that pointers don't "reach through" other pointers. `p=head;` makes `p` point to the first item. When we change `p` later, it won't affect `head`.
- Likewise, `p=p.next;` simply makes `p` move down one link in the chain. In this picture, `p` is the node with 7 and `p.next` is the node with 4 (it's really the arrow from 7 to 4, but what matters is it's pointing to the 4):

  ```
  L1--> 2--> 7--> 4--> 9-->null
             A
             |
             p
  ```

  After `p=p.next;`, `p` is aimed at the node with 4. It just slides down one node.
- `p` will eventually go past the last node and be `null`. That's fine. The loop checks for `null`, which is what you're supposed to do.

  If you think about it, regular array loops also go past the end, so this is the same.

We can use a pointer loop to count how many items are in the list. It's not much more complicated than printing them:

```
int getLength(IntNode head) {
  int len=0;
  for(IntNode p=head; p!=null; p=p.next) len++;
  return len;
}
```

As a quick check, suppose `L1` is `null`. When we call `myLen=getLength(L1);` then `p` starts as null, the loop won't run, and it returns 0, which is correct.

Another simple pointer loop is finding the largest item. As usual, I'll start out with the 1st as the largest, then have the loop start at the second:

```
int largest(IntNode head) {
  if(head==null) return -9999; // list is empty
  int largest=head.val; // 1st item is largest, for now
  for(IntNode p=head.next; p!=null; p=p.next)
    if(p.val>largest) largest=p.val;
  return largest;
}
```

The most interesting thing is how much it's like the array version of largest (once you get used to the different loop.)

Checking whether something is in the list is another simple pointer loop. For fun I'll use the semi-slimy shortcut where we reuse the input as the loop variable (that's why the 1st part of the `for` is blank):

```
bool isIn(IntNode head, int findMe) {
  for(;head!=null; head=head.next)
    if(head.val==findMe) return true;
  return false;
}
```

If this was an array, we'd rather find the position of the item, or -1 if it's not there. For a linked list we'd rather find the node with that item (or `null` if it's not there):

```
IntNode findNodeWithVal(IntNode head, int findMe) {
  for(IntNode p=head; p!=null; p=p.next)
    if(p.val==findMe) return p;
  return null;
}
```

If you don't trust the return-from-middle trick we can rewrite (this is just playing with loop conditions, which is always fun):

```
IntNode findNodeWithVal(IntNode head, int findMe) {
  for(IntNode p=head; p!=null && p.val!=findMel; p=p.next) {}
  return p; // null == not found
}
```

The other way to search is for a certain position. In arrays this is so easy (`A[4]`) that we don't even think about it. In a linked list we have to count that far from the start.

It's a fun loop, and a little tricky – we're still moving `p` but we're also counting. We don't want to make `p` be null, since it means the position was past the end, but we have to check just in case:

```
IntNode getNodeAtIndex(IntNode head, int wantIndex) {
  int i=0;
  IntNode p=head;
  while(i<wantIndex && p!=null) { p=p.next; i++; }
  return p;
  // NOTE: if index was too big, this returns null, which is the right answer
}
```

We'd use it like `IntNode pp = getNodeAtIndex(L1,4);`. To get the value, we'd use `pp.val`.

This next one is very sneaky – it uses the returned node. I want to count how many 7's are in a list. The plan is to use `findNodeWithVal` to find the first 7, count it, go to the next node, use `findNodeWithVal` starting from there to find the next 7 ... until we run out of list:

```
// count how many 7's there are in L1:
int count=0;
IntNode p=L1; // start looking at start of the list
while(p!=null) {
  p=findNodeWithVal(p, 7); // moves p to next 7 node, or null
  if(p!=null) { count++; p=p.next; }
}
```

We've seen the same idea in an array loop, except we keep searching and increasing `i`.

Fun fact: if we forgot `p=p.next;` after finding a 7, this would be an infinite loop, finding the same 7 over and over.

Another fun fact about all linked list loops: suppose the last thing in the list didn't point to `null`. Instead it pointed back to some previous node (the first one, somewhere in the middle, or even to itself.) Then loops over it would run forever, thinking the list was infinite.

35.1.2 Insert/Remove

The advantage of a linked list is we can easily insert an item into any position. We just splice it in by changing a few pointers. For example, here's a picture adding 55 to the start of our old list:

```
L1  2--> 7--> 4--> 9-->null
  \ |
   55
```

If it bothers you that memory is really numbered and is more like a line, here's a more memory-accurate picture of how splicing in 55 might look:

```
   --------------------
  /                    \
L1  2--> 7--> 4--> 9<>  55
     \                 /
      -----------------
```

To do this in code we need to create a new node, point it to the old first item, then make the head point to it:

```
void insertToFront(ref IntNode head, int newVal) {
  IntNode nn = new IntNode(); nn.val=newVal;
  nn.next=head; // point at old 1st item
  head=nn; // make this first item
}
```

We'd run this with `insertToFront(ref L1, 55);` (it needs to be passed by reference, since we're changing where `L1` points.)

Having to make a new node should make sense. If we have 4 items, we have 4 nodes. We don't need or want extra nodes hanging around. So if we want a new item, someone has to make the new node for it.

The order of the last two lines is important. If we started with `head=nn;` then we'd lose the list. Another way to write it would be:

```
// alternate hook-up code:
  IntNode oldFirst = head; // saved 1st item
  head=nn; // point to new 1st item...
  nn.next=oldFirst; // ...which points to old 1st item
}
```

One last thing to check – does this work inserting into an empty list? `L1` starts as `null`. This makes the new new point to `null`, so it works.

We can now finally create that 2, 7, 4, 9 list we've been using. Since we're adding to the front, we have to add them in reverse order:

```
IntNode L1=null;
insertToFront(9); // L1-->9
insertToFront(4); // L1-->4--9
insertToFront(7); // L1-->7-->4-->9
insertToFront(2); // L1-->2-->7-->4-->9
```

Inserting in any other position requires us to have the node before it. For example, inserting 15 after the 7 involves changing the `next` pointer from 7:

```
L1--> 2--> 7  4--> 9-->null
            \ |
             15
```

The code is pretty much the same:

```
void insertAfterNode(IntNode p, int newVal) {
  IntNode nn=new IntNode(); nn.val=newVal;
  nn.next=p.next; // new node points to the one after p
  p.next=nn; // p points to us
}
```

It's easy to get confused by the two `p.next`'s. Remember the trick with arrows is if it's on the left side, we only care about the the box where it starts, which we're changing. If it's on the right side, we only care about what it points to and not where it's coming from.

We could insert after the 7 like this:

```
// add a 15 after the 1st 7:
IntNode p = findNodeWithVal(L1, 7);
if(p!=null) insertAfterNode(p, 15);
```

Just for fun, here's a hacky way to add 15 to the end of the list, using our previous functions. Find the length, subtract 1, find that node, and insert after it:

```
int len=getLength(L1);
if(len==0) insertToFront(ref L1, 15);
else {
  IntNode last=getNodeAtIndex(len-1);
  insertAfterNode(last, 15);
}
```

Also just for fun, we could write a function that adds to the end a little nicer:

```
void insertAtEnd(ref IntNode head, int newVal) {
  IntNode nn=new IntNode(); nn.val=newVal;
  nn.next=null; // since it's the last item

  if(head==null) { head=nn; return; }
  // loop stops where we're _about_ to go off the edge:
  IntNode p=head;
  while(p.next!=null) p=p.next;
  p.next=nn;
}
```

Using `while(p.next!=null)` is a common trick to look ahead, stopping just before we go off the end. Another common way is keeping the normal loop that has `p` fall of the end, but adding an extra trailing variable:

```
// find last item using a trailer:
IntNode p=head, pPrev=null;
while(p!=null) {
  pPrev=p; // old value of p, just before moving it
  p=p.next;
}
// now p is off edge and pPrev is last node
```

This might look a little better, or not.

35.1.3 Removing

Removing an item is like inserting it. We just reroute the pointer around the item to remove. Like inserting, the first item is a special case, since we have to change `L1`.

Here's what it looks like to remove the first item of of 2, 7, 4, 9:

```
L1    2--> 7--> 4--> 9-->null
  \       /
   -------
```

It looks odd that 2 still points to 7. We could set it to `null`, but it won't matter. The key things are the list now starts with 7, then 4 then 9; and there's no way to get to 2 anymore. Eventually the garbage collector will remove it.

The code looks like this:

```
void removeFirst(ref IntNode head) {
  if(head==null) return; // there isn't a 1st item
  head=head.next;
}
```

As a check: if there's only 1 item, this would make `head` point to `null`, which is correct.

Removing something that's not the first item is just odd. We need the node *in front* of the one to remove. That's not very good (we'll design a better linked list to fix it, later.) Here's how it looks:

```
void removeAfter(IntNode beforeNode) {
  if(beforeNode.next==null) return; // nothing after us to remove
  beforeNode.next = beforeNode.next.next;
}
```

`beforeNode.next.next` means to follow the arrows twice. `beforeNode.next` is the one after us, so `(beforeNode.next).next` is two steps past (I added the parens, but it works the same either way.)

This last one is long and confusing, but it shows off pointers pretty well. We want to remove all negative numbers from a list. The loop will check whether the *next* number is negative. As usual, the first is a special case:

```
void removeNegatives(ref IntNode head) {
  if(head==null) return; // no items in list
  // removing 1st is special case, since we have to change head:
  // also, there might be several negatives at the start, so we need a loop!!:
  while(head.val<0) {
    head=head.next;
```

```
    if(head==null) return; // whole list was negative numbers
  }
  // check non-first items, looking 1 ahead:
  IntNode p=head;
  while(p.next!=null) { // while not last item
    if(p.next.val<0) p.next=p.next.next; // cut out next item
    else p=p.next; // move normally to next item
  }
}
```

This mess has all the linked list badness, and the usual deleting complications (the head is a special case; need to look-ahead one node so we can remove it; don't move forward when you delete something.)

Whenever I have to write something like this, it's always got a few little errors, including infinite loops and exciting null reference exceptions, and needs solid testing to shake them out.

35.2 Misc

For testing, it would be nice to make a linked list with just 0, 1, 2 ... in it. We'll use the trick where we insert them backwards into the front (the same way we made the 2,7,4,9 sample list):

```
IntList makeSeqList(int len) { // list from 0 to len-1
  IntList head=null;
  for(int i=len-1;i>=0;i--) // backwards, since adding to front
    insertToFront(ref head, i);
  return head;
}
```

For testing, it might also be nice to convert an array into a linked list. We can use the same trick – go through it backwards, adding to the front:

```
IntNode arrayToLinkList1() {
  IntNode head=null;
  for(int i=A.Length-1; i>=0; i--)
    insertToFront(ref head, A[i]);
  return head;
}
```

We'd use it like `IntNode L1=arrayToLinkList(new int[]{2,7,4,9})` (that's the C# way to write a constant array. Or we could just use `A`.)

Just for fun, what if we didn't solve this by thinking "adding to the front is easy" and working backwards from there? There are some other ways we could brute-force our way into this.

We might start with a regular array loop, adding to the end:

```
IntNode arrayToLinkList2() {
  IntNode head=null;
  for(int i=0; i<A.Length; i++)
    insertAtEnd(ref head, A[i]); // gah -- this is a loop
  return head;
}
```

We could rewrite that using an extra `tail` pointer to keep track of the end (we always call the last item in the list the tail.) That would get rid of the extra loops, but make it much uglier and harder to debug.

Another array-like way is we might imagine copying one array into another. First we'll make an "empty" linked list of the correct size. We'll use add-to-front, so can't put numbers in it yet. Then we'll loop through both together:

```
IntNode arrayToLinkList3(int[] A) {
  IntList head=null;
  // make enough "empty" nodes:
  for(int i=0;i<A.Length;i++)
    insertToFront(ref head, -1); // dummy value
  // now copy values:
  IntNode p=head; int i=0; // march these side-by-side
  while(i<A.Length) {
    p.val=A[i];
    i++; p=p.next; // 1 step ahead in both
  }
}
```

Reversing a linked list is another one that has a nice way if we think about linked lists, and some clumsy ways if we try to copy the array way of thinking.

Reversing an array looks like this:

```
// swap items in 1st and second half:
for(int i=0;i<A.Length/2;i++) // swap A[i] and A[A.Length-1-i]
```

We can brute force loop this to work with a linked list, but each function call is a loop, so this runs slow:

```
void reverse1(IntNode head) {
  int len=getLength(head);
  // swap 1st half with back half:
  for(int i=0;i<len/2;i++) {
    IntNode n1=getNodeAtIndex(head,i);
    IntNode n2=getNodeAtIndex(head,len-1-i);
    // standard swap:
    int temp=n1.val; n1.val=n2.val; n2.val=temp;
  }
}
```

The nicer version rearranges the nodes, something that requires a linked-list way of thinking. We go through the list, remove each node, and add it to the front of a new list (which will be reversed):

```
void reverse(ref IntNode head) { // head will change
  IntNode R=null; // temp holder for reverse list
  IntNode p=head; // loop variable
  while(p!=null) {
    IntNode savedNext=p.next;
    p.next=R; R=p; // insert p into the front of R
    p=savedNext;
  }
  head=R;
}
```

Notice how I couldn't use `insertToFront`. That takes a number and creates a node for it. In our case, we already have the node. We only want to change some pointers. The loop also had to save `p.next` at the top, before changing it.

If you recall, some functions change us, and some return changed copies of us (for example, `w.ToLower()`.) If we wanted a reversed *copy* we could do this:

```
IntNode makeReversedCopy(IntNode head) {
  IntNode A=null;
  for(IntNode p=head; p!=null; p=p.next)
    insertToFront(ref A, p.val);
  return A;
}
```

Another way to reverse a list starts with a linked-list-y idea, but isn't quite as nice when we write it. What if we flip every arrow in the picture? Make every node point to the one in before it, instead of after it:

```
void reverse3(ref IntNode head) {
  IntNode p=head; // loop variable
  IntNode prev=null; // trails behind p
  while(p!=null) {
    IntNode savedNext=p.next; // save next node
    p.next=prev; // switch p to point to the node behind it
    // now move the loop ahead a node:
    prev=p;
    p=savedNext;
  }
  head=prev; // when p is off end, prev is last node
}
```

This ends up being the same as the insert-to-front way, except `R` is replaced by `prev`, and we're thinking of it differently.

We can be more extreme with tearing apart a linked list. For example, we can split the list into even/odd:

```
IntNode odds=null, evens=null;
IntNode p=L1;
for(p!=null) {
  IntNode savedNext=p.next;
  if(p.val%2==0) { p.next=evens; evens=p; }
  else  { p.next=odds; odds=p; }
  p=savedNext;
}
// frontAdd on evens and odds reversed them, fix it:
reverse2(ref evens); reverse2(ref odds);
```

35.3 Special cases

Linked-list loops are also fun because, as we've seen, they have a lot of special cases: 1st item, last item or only item. I skipped mentioning this for most examples, but for real we have to think about each one.

For examples, adding after a node: it's not possible for our new item to be the first one or only one, but it could be the last. It so happens the regular code works for adding the last item, but we still had to check.

Our look-ahead loop using `while(p.next!null)` will crash on an empty list (since `p` starts as null.) We had to handle that as a special case.

But the trick is to ignore special cases at first. Assume you're in the middle of a long list and write general purpose code to solve it. Then go back and think about each special case. Some will work anyway, some won't and need us to add extra `if`'s.

35.4 Doubly-linked lists

For real, we like to have each node point to the next one and the previous one, like this:

```
class IntNode {
  public int val;
  public IntNode prev; // node in front of us. null==1st node
  public IntNode next;
}
```

Here's the new picture. The top row is `next`, the bottom is `prev`:

```
head-->| |-->| |-->| |-->| |-->| |-->null
       |2|   |7|   |4|   |9|   |3|
null<--| |<--| |<--| |<--| |<--| |<--tail
```

Advantages are we can look through the list backwards, we can find both nodes around us for deleting, we can insert to either side of us, and if we really need to we can make a loop that can walk back and forth.

Obviously we still have to go through one item at a time.

The drawback is we have to change twice as many pointers when we move things around.

The other improvement we usually make is saving the head, tail and length in a class. Then we write all the functions as member functions. The new linked list class:

```
class IntList {
  public IntNode head=null, tail=null;
  public int len=0;
  // insert, remove ... as member functions:
  // remember they get to use head, tail and len for free
}
```

Now instead if having `L1` be a simple `IntNode` pointer, we need `IntList L1 = new IntList();`. We have `L1.head=null;`, pretty much the same as before.

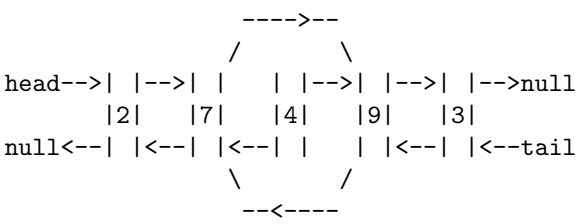
One fun thing we can do with this is search from either end. To find an index, we'll start from the back if it would be quicker:

```
// this is a member function:
public IntNode nodeAtIndex(int index) {
  if(index<0 || index>=len) return null; // off edge
  if(index<len/2) { // 1st half - search from front:
    IntNode p=head;
    for(int i=0;i<index;i++) p=p.next;
    return p;
  }
  else { // search from back:
    IntNode p=tail;
    for(int i=len-1;i>index;i--) p=p.prev;
    return p;
  }
}
```

We could do this before, but this way averages twice as fast. Plus it's fun.

Our removing ability from before was terrible, since we needed to know the previous node. Now if we have a pointer to a node, we can remove it.

Here's a picture how the two arrows change to remove the 4. Notice how the arrows from 4 still don't change, since they don't matter anyway:



If `nn` was pointing to the 4, these are the lines to change the arrow that way:

```
nn.prev.next=nn.next; // the node behind us points to node past us
nn.next.prev=nn.prev; // node past us points to node behind us
```

`nn.prev.next=` can be confusing. It's changing the `next` arrow of `nn.prev`. That's the top arrow in the picture.

The real remove member function needs to worry about removing the first and last, and needs to adjust the `len` variable:

```
public void removeNode(IntNode p) {
  // Adjust node in front of us. Special case if first item:
  if(p!=head) p.prev.next=p.next;
  else head=p.next;
  // Adjust node after us. Special case if last item:
  if(p!=tail) p.next.prev=p.prev;
  else tail=p.prev;
  len--;
}
```

A fun thing to do is trace this out for all the cases: middle, first, last, only node. It happens to work for them all (if it didn't, we'd fix with more `if`'s.)

Adding to the front is the same as before, except we have to update tail and len:

```
public void frontAdd(int newVal) {
  IntNode nn=new IntNode(); nn.val=newVal;
  nn.next=head; nn.prev=null; head=nn;
  // fix next node, or tail:
  if(len>0) nn.next.prev=nn;
  else tail=nn;
  len++;
}
```

Since we have the tail, we can quickly add to the back. It's the same as the front, except backwards, but it's still nice to see:

```
public void backAdd(int newVal) {
  // if list is empty, reuse frontAdd:
  if(len==0) { frontAdd(newVal); return; }
  IntNode nn=new IntNode(); nn.val=newVal;
  nn.prev=tail; nn.next=null; tail=nn;
  nn.prev.next=nn; // previous last node points to us
  len++;
}
```

Let me say again that these get confusing. I always have to draw a picture, recheck the code to be sure what `nn.prev.next` is, and hand-move the arrows.

Before, we could insert something after a node. Now we can choose to insert before or after. Here's insertAfter. We need to change 4 arrows total (the 2 from us and the 2 to us.) Inserting the last is still a special case:

```
public void insertAfter(IntNode p, int newVal) {
  if(p==tail) { backAdd(newVal); return; }
  IntNode nn=new IntNode(); nn.val=newVal;
  nn.next=p.next; nn.prev=p; // our arrows
  p.next=nn;
  nn.next.prev=nn;
  len++;
}
```

We can be really sneaky writing `insertBefore`, by going back a node and reusing insertAfter:

```
public void insertBefore(IntNode p, int newVal) {
  if(p==head) { frontAdd(newVal); return; }
  insertAfter(p.prev, newVal);
}
```

The other things we could write are frontRemove and backRemove. Searching for a particular item is the same as before. Reversing is still front-adding into a new list. Array-to-list is still backwards through the array and using front adds.

There are a few very, very fancy version of linked lists with even more pointers, but you rarely use them and the math is hard. Just a next and prev pointer is a standard grown-up linked list.

35.5 Using templates

This section is just about using templates, and not really about linked lists at all. Except almost every built-in linked list uses templates.

We could make a different linked list node for strings, floats, Cats ..., but that would be a pain. Switching what they hold is why template classes were invented (C# renamed them *generics*.)

Our node type will look like:

```
class Node<T> {
  public T val;
  public Node<T> next, prev;
}
```

Remember the first `Node<T>` defines us as requiring an extra input. It says the real classes will be `Node<int>` or `Node<string>` or even `Node<Cat[]>` (each item in the list is an array of Cats.)

Then the `T's` inside automatically get filled in. If we create a `Node<string> nn = new Node<string>();`, then the next and prev fields are automatically also `Node<string>`'s.

The base list class would be:

```
class LinkList<T> {
  public Node<T> head=null, tail=null;
  int len=0;

  public addFirst(T newVal) {
    Node<T> nn = new Node<T>(); nn.val=newVal;
    nn.next=head; head=nn;
    ...
  }
}
```

We'd use this with `LinkList<string> L1 = new LinkList<string>();`

The inside `T`'s are also filled in with `string`'s. For example, calling `L1.addFirst("cow");` knows to create a new `Node<string>` for our cow.

35.6 Built in Linked List

Most languages have built-in doubly linked lists that work about the same. C#'s is nothing special, but it's interesting to see how they change a few things.

It's obviously a template class. The node type is named `LinkedListNode<T>`. You'll only use this for loops or as the result of a search. The whole type is

`LinkedList<T>`. The member variables are private, but are obviously `head`, `tail` and a length.

The functions to manipulate them are the same ones we've been using, named a little differently:

```
LinkedList<string> L1 = new LinkedList<string>();
L1.AddFirst("bb"); L1.AddFirst("aa");
L1.AddLast("ccc"); // aa, bb, ccc

LinkedListNode<string> nn = L1.Find("bb");
L1.AddBefore(nn,"a2"); // aa, a2, bb, ccc
L1.Remove(nn); // aa, a2, ccc
```

You can't see the head or tail directly, but you can use `L1.First` and `L1.Last` to get them (why not call these head and tail? C# is partly designed for entry-level programmers who aren't used to those terms.)

Since everything is private, you can't directly change the pointers, but, for example, you can still tear a list apart using built-in commands. This pulls all the odd nodes out of `L2`, moving them into another one:

```
LinkedList<int> Odds = new LinkedList<int>();
LinkedListNode<int> p = L2.First; // loop pointer
while(p!=null) {
  LinkedListNode<int> savedNext = p.Next;
  if(p.Value%2==1) {
    L2.Remove(p);
    Odds.AddFirst(p); // overloaded to let you add a node
  }
  p=savedNext;
}
```

35.7 Array implementation of linked lists

This section isn't very useful, but it's more practice for thinking about how linked list work and more playing with array indexes. It is a real thing, but it's only used in very specific circumstances.

The trick is, we pre-make an array with all of nodes we'll ever use. Each `next` pointer will be the index of the next item. The `head` pointer is the index of the first item.

This negates a big feature of linked lists – we won't be able to grow it all we want. But we can still do the other nice stuff: insert and remove from anywhere.

The nodes will look like this:

```
struct Node {
  public string val; // using strings to be less confusing
  public int next; // index in array of next item. 0 to length-1
}
```

Then we'd create an array making all the nodes we think we'll ever need: `Node[] AllNodes = new Node[10];`. In our minds, this is ten unused nodes. For now, suppose `next=-999;` marks it as unused.

We'll start `int L1=-1;`. It holds the 0-9 index of the first node in the list, with -1 for null.

The spots would be probably be used in order, but if we kept adding and removing we might get something like this. The list is boxes 3, 5, 2, 0, 7. Each box has the number of the next one:

```
L1: 3

   0     1     2     3     4     5     6     7     8
 ----- ----- ----- ----- ----- ----- ----- ----- -----
| ddd |     | ccc | aaa |     | bbb |     | eee |     |
|  7  |     |  0  |  5  |     |  2  |     | -1  |     |
 ----- ----- ----- ----- ----- ----- ----- ----- -----
```

A loop to print them out looks a lot like a standard pointer loop. `p` will jump through 3, 5, 2, 0, 7 then -1:

```
for(int p=L1; p!=-1; p=AllNodes[p].next)
  print(AllNodes[p].val);
```

Inserting to the front is similar. Pretend we have a function to get an unused node:

```
int nn = getUnusedIndex(); // 0-9 of a free node
AllNodes[nn].next=L1; L1=nn;
```

Deleting the node after `p` should also look similar:

```
int p2 = AllNodes[p].next;
int p3 = AllNodes[AllNodes[p].next].next; // ugg
AllNode[p].next=p3; // skip past
AllNodes[p2].next=-999; // mark as unused
```

These examples are singly linked, but it works doubly-linked as well.