Chapter 42

Linked Lists

A linked list is an alternative to an array. It stores a list of items, and can't do anything an array can't do. We care about them because they're very fast at inserting & removing items, from anywhere, and arrays are very, very slow at that.

We don't usually care about a little speed here and there, but choosing the wrong type of list (array or linked list) can give a x1000 slowdown. So linked lists are one of the basic data structure all programmers learn.

Linked lists use pointers, as real pointers, a lot. Playing with linked lists to get better with pointers is very traditional. This chapter also has some fun functions doing that.

A note: we've also seen `C#`'s `List<int>` class. If you remember, that's just a class holding an array. Some languages have lots of alternate data types for storing lists. Deep down, they're always either an array, or a linked list.

42.1 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.)

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
}
```

The picture above has 4 `intNode`'s, and `L1` is of type `intNode` (but it's only a pointer to the first one.)

Our picture written out as IntNodes would look like this:

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

Backing up a little, if you look at the second field, `public IntNode next;`, you might think "wait – we have an IntNode inside of another IntNode?" Well, no.

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 where 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 being used as a real pointer. Each IntNode "knows about" the one after it, using `next`.

This is also the first good example of how `new` creates "nameless" heap objects. Every time before this, we would have needed `L2`, `L3` and `L4` variables to point to the other three actual nodes.

Now we think of `L1` as being the whole list. The 4 intNode's feel more like just floating things with no specific names. For the 9 node, none of our declared variables points to it, and that's fine. The only way to find it is by following the chain.

42.1.1 Linked List loops

For now, let's just pretend we can make the chain of nodes somehow, and jump straight to using 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
  }
}
```

It walks `p` to point at each item, until it goes off the end. 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) // loop from 2nd item
    if(p.val>largest) largest=p.val;
  return largest;
}
```

Notice the way the `for` loop starts at the second item: `intNode p=head.next`. This is roughly the same as `int i=1;` in an array 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;
}
```

As we know, instead of saying true/false if something is in an array, we'd rather return the position, or -1. For a linked list, positions aren't as useful. We'd rather return a pointer to the node, or `null`. This revised item-finding loop does that:

```
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!=findMe; p=p.next) {}
  return p; // null == not found
}
```

The {}'s after the `for` loop say what they do: walk through the loop until you find the value or `null`, and for each item, do nothing.

The thing linked lists are terrible at, is jumping to a certain position. In an array, `A[5]` jumps straight to the fifth item. Finding the Nth item in a linked list requires a loop.

It's a little tricky since we're moving and counting and need to check for falling off the end (if the list isn't that long). Also, instead of returning just the Nth number, we'd rather return a pointer to that node:

```
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`.

Counting how many 7's are in a linked-list could be another "check every box" loop. But there's a much more fun way.

The plan is to use `findNodeWithVal` over and over, always starting 1 past where the last seven was, 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) { // found one
    count++; // count it
    p=p.next; // next search starts at next node
  }
}
```

This is a loop within a loop: `findNodeWithVal` skips to one 7, and the loop we wrote makes it keep finding more. We've done this before with arrays. With them, we used `int i;` and `i+1` to go to the next. With a linked list the same thing is `p` and `p.next`.

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

42.1.2 Insert/Remove

Everything above would have been easier with an array. Linked Lists big advantage is inserting and removing items.

With an array, almost every add/remove requires a loop to slide everything to fill the empty spot. With a linked list, we can easily insert and remove from anywhere, as long as we have a pointer to a nearby node.

To insert, we splice it in by changing 2 pointers 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
     \                 /
      -----------------
```

Following those long arrows is a pain for us, but it's not work at all for the computer. Following any arrow is exactly as easy as any other.

The code is what you see in the picture: create a new node, point it to the former first item, make the head point to the new item. This takes the start of the list as a reference parameter (since the function needs to change it):

```
void insertToFront(ref IntNode head, int newVal) {
  IntNode newNode = new IntNode(); newNode.val=newVal; // make the node
  // hook it up:
  newNode.next=head; // new one points at old 1st item
  head=newNode; // make this first item
}
```

Calling `insertToFront(ref L1, 55);` would make our new picture.

Hooking it up is a little tricky. If we flipped the order of the last two lines, it wouldn't work (`head=newNode` would lose our only link to the list.)

Another way to hook it up would with an extra pointer:

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

One last thing to check – does this work inserting into an empty list? `L1` would start out as `null`. Then it would point to the 55, which would point to `null`, so it works (in the intro, when I wrote these were good practice with pointers, this is what I meant).

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
```

But besides that it's the same 2-pointer dance:

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

`newNode.next = p.next` is probably a big "yikes!" To read it, `newNode.next=` means we're changing where 15 points. `=p.next` is the new target, which is the node after 7. The whole line means "make the 15 point to the 4" (no one has ever understood that on the first try, just so you know. I haven't shown you a single new rule, but this is in the advanced chapters for a reason.)

To use it, and this is pretty cool, we could run our function that finds the 7. It returns a pointer to that node, which is exactly what insert needs:

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

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. We'd never do this for real, but it's a nice exercise:

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

This is still messing around: we could write a function that does every step by hand:

```
void insertAtEnd(ref IntNode head, int newVal) {
  // pre-make the new node:
  IntNode newNode=new IntNode(); newNode.val=newVal;
  newNode.next=null; // since it's the last item

  if(head==null) { head=newNode; return; }
  // loop stops where we're _about_ to go off the edge:
  IntNode p=head;
  while(p.next!=null) // <- if NEXT item is null. Tricky
    p=p.next;
  // p is last node. Add us after it:
  p.next=nn;
}
```

`while(p.next!=null)` is one of those extra-sneaky pointer linked-list tricks. When the next item is `null`, it means we're on the last item.

Another trick for finding the last item is adding a trailing variable. Let `p` fall of the edge, as normal:

```
// find last item using a trailer:
IntNode p=head;
IntNode pPrev=null; // trails behind p by 1 node
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
```

You may be thinking that the thing before `i` in an array is `i-1`, which is so much easier. But all of this extra work with linked lists is worth it for the speed if we need to frequently insert and remove from anywhere.

42.1.3 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 anything else requires the node in front of it. We again route the pointer around

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

`beforeNode.next.next` isn't a joke, or magic. It's the pointer to 2 nodes after you. Read it as if it had parens: `(beforeNode.next).next` If `beforeNode` is the 2, it jumps to the 7, then checks the `next` in there, which is an arrow to the 2.

But, again, yikes! Experienced coders are able to sight-read this, but it takes lots of practice.

This last one is a loop to remove all negative numbers from a linked list. Since we can only remove the next number, the loop checks whether the next number is negative. As usual, removing the first is a special case:

```
void removeAllNegatives(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; // skip past = remove
    if(head==null) return; // whole list was negative numbers
  }
  // check non-first items, looking 1 ahead:
  IntNode p=head; // NOTE: we know this is not null
  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
  }
}
```

There are so many way this could go wrong. But it's typical linked-list pointer math.

The sneakist part is that if we remove something, we *don't* move forward. The removal puts a fresh item in front of us. Also moving would jump past it – we'd have a bug when there were 2 negative numbers in a row.

42.2 Misc

For testing, it would be nice to make a linked list with 0 to 9. This does it:

```
IntNode makeSeqList(int len) { // list from 0 to len-1
  IntNode head=null;
  for(int i=len-1;i>=0;i--) // backwards, since adding to front
    insertToFront(ref head, i); // inserts 9, 8 ... 0
  return head;
}
```

`IntNode L2 = makeSeqList(10);` runs it.

It might also be nice to convert an array into a linked list. We can go through the array backwards, adding each to the front:

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

`IntNode L3=arrayToLinkList(new int[]{2,7,4,9})` would run it (or use an array you've already made).

Suppose we didn't know adding to the front is fast and the end is slow. We could write array-to-list this way, which seems simpler:

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

If `A` has 100 items, which isn't big at all, this takes 1+2+3 ... + 100 steps – – roughly 5,000, instead of only 100 if we used front-insert.

Another plan, which isn't as good but seems more obvious, is to make a Linked List of empty nodes, then copy `A` into them:

```
IntNode arrayToLinkList3(int[] A) {
  // make list with all 0's, same length as A:
  IntList head=null;
  for(int i=0;i<A.Length;i++) insertToFront(ref head, 0);
  // 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.

Array thinking is to swap values from the first and last, then second and second-to-last, as so on:

```
void reverseArrayStyle(IntNode head) {
  int len=getLength(head);
  // position of the 2 to swap. Move these inward, together:
  int i1=0, i2=len-1;
  while(i1<i2) {
    IntNode n1=getNodeAtIndex(head, i1); // <- loop
    IntNode n2=getNodeAtIndex(head, i2); // ditto
    // standard swap:
    int temp=n1.val; n1.val=n2.val; n2.val=temp;
    i1++; i2--;
  }
}
```

That works, and the logic is clear – we're used to it from arrays. But it's a nested loop. 10,000 steps to reverse a length 100 list.

The much faster version uses linked-list thinking to rearrange the nodes (which is impossible for an array). Remove nodes from the front, and add it to the front of a new list. The new one is backwards, because that's how insert-to-front works:

```
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; // next in the starting array
  }
  head=R; // change original list pointer to fixed list
}
```

That's another linked-list loop which is much trickier than how short it is.

If you recall, we often don't like functions that change us. We prefer ones that return a copy with things the way we wanted. This returns a copy of `L`, but reversed (this is our first list-copying function):

```
IntNode makeReversedCopy(IntNode L) {
  IntNode A=null; // answer (L reversed)
  for(IntNode p=head; p!=null; p=p.next) // standard list loop
    insertToFront(ref A, p.val);
  return A;
}
```

Another list-y plan is to reverse every arrow. We'll go through front-to-back, using a trailing pointer, flipping arrows as we go (the arrow from `prev` to `p` is flipped to go from `p` back to `prev`). This isn't as good, and is too complicated:

```
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 seems harder to read than the insert-to-front method, and isn't any faster. But it's fine for an exercise.

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; // heads of 2 lists
IntNode p=L1;  // L1 is the original list
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;
}
// problem: both lists were backwards, since we front-added. Fix it:
reverse2(ref evens); reverse2(ref odds);
```

42.3 Special cases

You may have noticed that about half of each chunk of code is dealing with the list being empty, and handling the first node, which is a special case.

That's typical of lots of things.

The trick is writing a good function is ignoring special cases at first. For these, assume you've for a nice, long list and pretend the loop is about halfway through. Write code to keep going from there.

Then think about to start the loop. Then worry about empty lists, only length 1 lists and any other funny stuff you can think of.

42.4 Doubly-linked lists

The final form of a real linked-list is giving each node an extra pointer to the node in front of it. This is known as a Double-Linked List.

We usually call the second pointer `prev`, for *previous*. The new Node class looks 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`. Since we can walk backwards now, we also save a pointer to the end, named `tail`:

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

An advantage is being able to remove a Node based on its node pointer. Another is being able to go through the list backwards, and a minor one is being able to insert to either side of a Node.

But this won't change the basic linked list nature: insert/remove is fast; but searching for item N is slow.

We also like to make a nice class to hold the head and tail, plus the length, since why not. Then we can write everything as member functions. The new linked list class:

```
class IntList {
  public IntNode head=null, tail=null;
  public int len=0; // member functions need to keep this updated
  // insert, remove ... as member functions:
  // remember they can all see head, tail and len
}
```

Now we use `IntList L1 = new IntList();` to make one, and `L1.frontAdd(6);` to insert a new item.

Here's the new add-to-front member function. Setting up both forward pointers is the same; and it also need to set up the two backwards pointers:

```
public void frontAdd(int newVal) {
  IntNode newNode=new IntNode(); newNode.val=newVal;
  newNode.next=head; newNode.prev=null;
  head=newNode;
  // the old first node needs to point back to us:
  if(len>0) newNode.next.prev=newNode;
  else tail=newNode;
  len++;
}
```

`newNode.next.prev=newNode;` is another brain-teaser. Reading it with parens: `(newNode.next).prev=`. It finds the node after the new one, and changes its `prev` pointer to aim back to the new one. This is one of the most confusing lines in pointer math.

Also notice how inserting to an empty list is a special case (there's no node to point backwards to the new one, but the main `tail` should point to it.)

Since we have the tail, we can easily add 1 new item to the back. The code is (obviously) the same as adding to the front, except reversed:

```
public void backAdd(int newVal) {
```

```
  // if list is empty, it's easier to reuse frontAdd:
  if(len==0) { frontAdd(newVal); return; }

  IntNode newNode=new IntNode(); newNode.val=newVal;
  newNode.prev=tail; newNode.next=null;
  tail=newNode;
  newNode.prev.next=newNode; // previous last node points to us
  len++;
}
```

Now, with confidence, we can use `L1.backAdd(7);`.

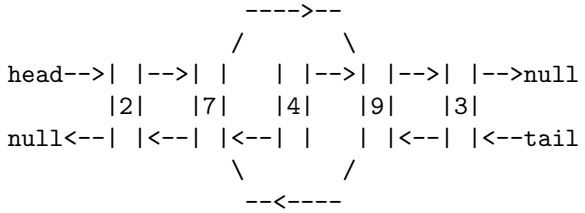
This next one is just fun. Finding an item at a particular index is still a slow loop, but we can start from the back, if that would be faster:

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

Hopefully the indexing for the backwards walk looks OK. It's using the actual index of that item, which is why it subtracts. If we want to find item 36 in a length 50 list, the loop counts backwards from 49 until it hits 36.

Now that we know both nodes around us, we can remove a Node we know about (none of that needing the previous item nonsense, from before.)

To cut out the 4 in this picture, we route the `next` arrow around it, as usual, and also the `prev` arrow. Also the same as before, the arrows from 4 don't matter, since we can't get to 4 anymore:



Suppose `nn` is pointing to the 4, these lines change the arrows to cut it out:

```
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
```

They are both different versions of the hardest thing about pointers.

The real remove member function needs to worry about special cases, and adjusting the `len` variable:

```
public void removeNode(IntNode p) {
  if(p==null) return; // may as well check this
  // node in front of us goes around. Special case if first item:
  if(p!=head) p.prev.next=p.next;
  else head=p.next;
  // node after us goes around. Special case if last item:
  if(p!=tail) p.next.prev=p.prev;
  else tail=p.prev;
  len--;
}
```

As I wrote, doubly-linked lists let us insert after or before a Node. Here's after:

```
public void insertAfter(IntNode p, int newVal) {
  if(p==tail) { backAdd(newVal); return; } // after p is the end

  IntNode nn=new IntNode(); nn.val=newVal;
  nn.next=p.next; nn.prev=p; // our arrows
  p.next=nn;
  nn.next.prev=nn;
  len++;
}
```

To write `insertBefore` we can reuse `insertAfter`:

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

I'm skipping frontRemove and backRemove. They're more of the same. Searching for a particular item is also no change.

42.5 An any-type linked-list, using templates

If you remember, the built-in "nice array" looks like `List<int> L=new List<int>();`. using templates. This section doesn't have any new linked-list stuff in it – only using templates to make them nicer.

Our Node class now holds type `T`:

```
class Node<T> {
  public T val; // T is whatever type the list holds
  public Node<T> next, prev;
}
```

Remember the `<T>` in `Node<T>` means we have to supply a type, and the `T val;` inside says to replace `T` with int or string or whatever they supplied.

The base list class also gains a `<T>`. Notice how it uses `T` to define the Node's it uses:

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

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

We'd use `LinkList<string> L1 = new LinkList<string>();`. Then `L1.frontAdd("frog");`.

Member function `frontAdd` gets to use `(T newVal)` as the input since it remembers whatever `<T>` we used when we created `L1`. That way `L2.frontAdd(6.4f);` could be legal, since it uses `L2`'s value for `<T>`. This is all template sneakiness.

Everything else is the same, except they use `T`'s instead of `int`'s. I think this is pretty neat, since we can hand-code, without too much trouble, an actually useful linked list class.

42.6 Built in Linked List

As you'd guess, `C#` has a built-in Linked List class. The variables are all private, but we use the functions the same way. For example:

```
LinkedList<string> L1 = new LinkedList<string>(); // empty
// add to front or back:
```

```
L1.AddFirst("bb"); L1.AddFirst("aa");
L1.AddLast("ccc"); // aa, bb, ccc

// searching returns a NODE pointer:
LinkedListNode<string> nn = L1.Find("bb");
string w = nn.Value; // Value instead of just val

// can add items before or after a node:
L1.AddBefore(nn,"a2"); // aa, a2, bb, ccc
L1.AddAfter(nn,"a4"); // aa, a2, bb, a4, ccc

// remove takes a node as input:
L1.Remove(nn); // aa, a2, a4, ccc
```

You can't see the head or tail directly, but you can use `L1.First` and `L1.Last` to get those pointers (to Nodes). Those names are kind of interesting. Someone who took a course would prefer `head` and `tail`, since it reminds them it's a linked list. But that would confuse people who didn't take a course.

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.AddLast(p); // has a version to add a node
  }
  p=savedNext;
}
```

This sort of thing is rare. I like it since you can use the official linked-lists with simple Add and Remove commands. But if you know how they really work, the system gives you access to the internal nodes and lets you shift them around.

42.7 Array implementation of linked lists

This section isn't very useful, but it's more practice for thinking about how linked lists 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 make `Node[] AllNodes = new Node[10];`. In our minds, this is ten unused nodes.

Here's a picture of a list with: aaa, bbb, ccc, ddd, eee:

```
L1: 3

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

Notice how our head, `L3`, points to box 3, with `aaa`. That box points to box 5, with `bbb`. At the end, box 7 uses -1 for null.

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.