Algorithms on Linked Lists

Wayne Snyder

CS 112, Spring 2013

These notes collect together a number of important algorithms operating on linked lists. They are intended as a supplement to the material in the textbook on p.142 and following. We also include some basic information about recursive algorithms. In particular, the topics we cover are as follows:

1. Basic (iterative) algorithms on singly-linked lists;
2. An introduction to recursion [Optional]; and
3. Recursive algorithms on SLLs.

The algorithms are generally presented as Java functions (methods) operating on lists of integers, however, note that these algorithms are only the minimal specification of what would be used in a practical situation. Your text presents these as generic data structures, and this is of course the more appropriate way to write real applications of thes principles. You would need to embed these in the appropriate classes to accomplish a specific purpose.

Iterative Algorithms on Linked Lists

Basic data declarations

All the code below assumes the following declaration:

```
public class Node {

    public int item;
    public Node next;

    Node() {                 // this would be the default, put here for reference
       item = 0;
       next = null;
    }

    Node(int n) {
       item = n;
       next = null;
    }

    Node(int n, Node p) {
       item = n;
       next = p;
    }
};
```

We will assume that there is a variable which points to the head of the list:

```
Node head;
```

Note that references in Java are initialized to **`null`** by default. These constructors will simplify a number of the algorithms below. For example, to create a list with one element containing the item 5, we could write:

```
head = new Node(5);
```

To add a node containing a 7 to the front of an existing list, we could simply write

```
head = new Node(7, list);
```

and if we have a reference `p` to a node in a list, we can add a new node containing 13 after `p` (i.e., between the node `p` and the node `p.next`) by writing:

```
p.next = new Node(13, p.next);
```

Note that it can also be used to create simple linked lists in a single Java statement. For example, to create a list containing the integers 7, 8, and 12, we could write:

```
head = new Node( 7, new Node( 8, new Node( 12 ) ) )
```

Basic Paradigms for Chaining down a List

Basic chaining (example: printing out a list)

The basic thing you do with a list is to "chain along" the list, setting a pointer `p` to each node in turn and performing some operation at each node (such as printing out the list). This is done with a simple `for` loop that initializes a reference to the first node and stops when it reaches null. Let's take printing out all the member of a list as an example.

The basic pattern is that this will execute once for each node in the list, pointing `p` at each node in turn:

```
for(Node p = head; p != null; p = p.next ) {
  System.out.println(p.item);
}

 // OR:

Node p = head;
while( p != null ) {
  System.out.println(p.item);
  p = p.next;
}
```

Written as a self-contained method taking the list as a parameter (the usual case), this would be:

```
void printList(Node p) {
  for( ; p!=null; p = p.next) {
    System.out.println(p.item);
  }
}
```

The basic pattern is that this will will execute once for each node in the list, pointing `p` at each node in turn:

```
for(Node p = head; p != null; p = p.next ) {
   // Do something at each node in the list
}
```

(Note we are guaranteed by the loop condition that `p` is not `null`, so we can refer to `p.item` or `p.next` anytime we want inside the loop without worrying about `NullPointerExceptions`.)

Another simple example would be finding the length of a list, which can be done by simply incrementing a counter inside the for loop:

```
int count = 0;
for(Node p = head; p != null; p = p.next ) {
   ++count;
}
// now count contains the number of items in the list
```

Chaining down the list and stopping at the end or upon some condition

Suppose we wish to find the first node in the list satisfying some condition C (e.g., we are looking for a particular item k, so C would be `p.item == k`). A simple modification of the previous technique is to add an `if` statement to the loop, possibly jumping out of the loop after executing some statements:

```
for(Node p = head; p != null; p = p.next )
   if( C ) {
      // do something to the node that p refers to
      break;   // if you want to stop the chaining along; or one could set p = null
   }
```

Note that we can refer to `p.item` and `p.next` in the loop body with no worry of a `NullPointerException`, because we have already checked that `p != null` in the for loop condition.

Basic Paradigms for Modifying a List (Example: deleting a node)

Chaining down a list is often combined with some kind of modification of the structure of the list, for example, you might want to insert or delete a node. We will consider node deletion as an example to illustrate the basic problem with a singly-linked list: a list goes in one direction, and you can go backwards! Suppose you want to delete the node pointed to by p in the following list:

```
head ->  2    ->     6   ->  9  ->  .
                      ^
                      |
                      p
```

The problem is that to delete the 6, you need to "reroute" the pointer in the node containing 2 so that it points to 9. But you don't have a pointer to this node! The simplest solution is just to keep a "trailing pointer" q that always points to the node right before p:

```
head ->  2    ->     6   ->  9  ->  .
         ^           ^
         |           |
         q           p
```

Now, to delete the 6, we can simply perform the following to "reroute" the next pointer in q to point to the node after p:

```
q.next = p.next;
```

This general technique is sometimes called the "inchworm" since the movement of the two pointers is like the way an inchworm moves its body.

```
Node  p = head;
Node q = null;
while( p != null ) {
                                  // During first iteration, p points to first element and q has no value;
                                  // thereafter, p points to a node and q points to the previous node
   q = p;
   p = p.next;
}

// OR, using a for loop:

for(Node  p = head, Node q = null; p != null; q = p, p = p.next ) {
   // During first iteration, p points to first element and q has no value;
   // thereafter, p points to a node and q points to the previous node
}
```

But it is a little bit complicated! What if we want to delete the first node? What if the list is empty? These are the special cases that make iterative algorithms a bit messy (we'll introduce a technique using "header nodes" to avoid this below). For example, to delete the first node in the list containing a negative number, we could do the following:

```
if( head == null)             // Case 1: List is empty, do nothing
   ;
else if( head.item < 0 )      // Case 2: have to delete first node?
   head = head.next;          //    Yes, so reroute around the first node
else {                        // Case 3: Don't have to delete first node, so can use inchworm

  Node p = head.next;         // p points to second node
  Node q = head;              // q points to first node
  while( p != null ) {
     if( p.item < 0 ) {
       q.next = p.next;
       break;
     }
     q = p;                   // chain along, and keep q trailing p
     p = p.next;
  }
}

// OR, using a for loop:

if( head == null)              // Case 1: List is empty, do nothing
   ;
else if( head.item < 0 )      // Case 2: have to delete first node?
   head = head.next;          //    Yes, so reroute around the first node
else {                        // Case 3: Don't have to delete first node, so can use inchworm

  for(Node p = head, Node q = null; p != null; q = p, p = p.next ) {
     if( p.item < 0 ) {
       q.next = p.next;
       break;
     }
  }
}
```

In general, the Inchworm technique is the easiest way to manage this problem, so we will use it in the rest of these notes. However, be careful about those special cases! In general, we have to worry about the following cases when modifying a linked list:

1. The list is empty;
2. The modification takes place at the beginning of the list (e.g., you are inserting a node at the front, or deleting the first node);
3. The modification takes place at the end of the list; and
4. The modification takes place somewhere in the middle.

Deleting a node involves cases 1 - 3, as we just saw; when inserting we have to worry about all four, as we will see below.

Iterative Algorithms for Linked Lists

We now present a "cookbook" of a number of useful iterative algorithms. Other algorithms can usually be created by suitably modifying one of the ones found here.

Print the List

```
void printList(Node h)  {
    for(Node p = h ; p != null; p = p.next )
```

```
            System.out.println(p.item);
     }
```

You would call the method like this:

```
    printList(head);
```

Finding the length of a list

```
  int length(Node h) {
    int count = 0;
    for(Node p = h ; p!=null; p = p.next)
      ++count;
    return count;
  }

// You would call the method like this:

    int n = length(head);
```

Looking up a item

This next function is a standard one; it returns a reference to the node containing n if it exists, and null otherwise:

```
  Node lookup(Node h,  int n) {
       for(Node p = h; p != null; p = p.next )
          if( p.item == n )
              return p;
       return null;             // return null if item not found
    }

// You would call the method like this:

    Node q = lookup(head, 23);
```

Deleting an item in a list

Here is a version of delete which removes the first instance of a specific number from the list; as observed above, the complication is that we have a number of special cases, depending on where the node to be deleted is. We have to have access to the head variable inside the method, in case we are deleting the first node, so you would have to write a method like this for each list (i.e., you can't just write a static method to be used on all such lists).

```
   void delete(int n ) {
       if(head == null)           // case 1: list is empty, do nothing
           ;
       else if(head.item == n)  // case 2: n occurs in first node
          head = head.next;       //    skip around first node
       else {                     // case 3: find the node before n using the inchworm technique

         for(Node p = head.next, q = head; p != null ; q = p, p = p.next ) {
           if( p.item == n ) {
              q.next = p.next;
              return;              // if you delete this line it will remove all instances of the number
           }
         }
       }
     }

// You would call the method like this:

    delete(23);
```

Inserting an item into a sorted list

We have shown above how easy it is to insert an item into the first position using the `Node()` constructor; inserting into a sorted list (in ascending order) is quite similar to the delete algorithm and again uses the two-pointer technique. Again, you would have to write a separate algorithm for each list, since you might have to modify the head pointer. In this algorithm we will use a while loop just to show how that works, as compared with the for loop.

```
  void  insertInOrder( int n ) {
       if(head == null)              // case 1: list is empty
          head = new Node(n);
       else if(head.item >= n)     // case 2: n should be before the first node
          head = new Node(n, head);        //    push on front of list
       else {                         // case 3: find the node before where n should be using the inchworm technique

        Node p = head.next;
        Node q = head;
        while( p != null ) {
           if(p.item >= n) {               // found insertion point, between p and q
                q.next = new Node(n, p);
                return;
```

```
            }
            q = p;                      // chain along!
            p = p.next;
         }
         q.next = new Node(n, null);   // case 4: Node must be added at end of list
        }
  }

// You would call this as follows:

     insertInOrder(23);
```

Copying a list

This algorithm is an example of one that is fairly messy in the iterative case, but almost trivial in the recursive case; we include it here so that you can compare it with the recursive one given later. It is also presented as an example of a technique that is very useful in maintaining a linked list, that is, maintaining a pointer to the last node in the list (e.g., if you want to continually add nodes to the end of the list, as here).

```
  Node copyList(Node p) {

    if(p == null)                   // Case 1: list is empty
      return null;
    else {                          // Case 2: list has at least one node
      Node c = new Node(p.item);    //    put first node in place in copy
      Node last = c;                // Maintain a pointer to the last node to facilitate adding to the end of the list
      p = p.next;
      while( p != null ) {
        last.next = new Node(p.item);
        last = last.next;           // chain along original list and with the last node in new list
        p = p.next;
      }
      return c;
    }
  }


// You would call this as follows:

Node q = copyList(head);
```

Reversing a list

This algorithm is probably one of the most difficult for linked lists; it uses three references which chain down the list together and rearrange the pointers so that node q, instead of pointing to p, now points to r; doing this for each node reverses the entire list. This was a standard exam question on the PhD Qualifying Exam at UPenn when I was a student there..... I've also heard that it is a common interview question at software companies!

```
    void reverse() {
        if(head == null)
            ;
        Node p = head.next,  q = head,  r;
        while ( p != null ) {
           r = q;         // r follows q
           q = p;         // q follows p
           p = p.next;   // p moves to next node
           q.next = r;   // link q to preceding node
        }
        head.next = null;
        head = q;
     }
```

Eliminating a Messy Special Case: Linked Lists with Header Nodes

The special cases 1 and 2 in these algorithms, when you need to worry about whether you need to change the head pointer, causes significant problems:

You have to write these special cases into the algorithms, as we have seen above; and

You must write a separate method for EACH linked list, as the pointer head must be explicitly mentioned in the algorithm.

This latter problem is really the most important, as it makes us write multiple versions of the same code, which, as we saw in the case of generics, is a real problem. We would like to write ONE method which can be used for all such lists. This will be solved elegantly when we use recursion, but we can solve the problem in the iterative case by using a **dummy first node** that contains no value (or a sentinel value, such as Integer.MIN_VALUE) as a item. In this case, you would initialize your list using:

```
        Node head = new Node();
```

Alternatively, and sometimes very usefully, we can use the header node to store useful information such as the length of the list. The general outline of a `for` loop which chains down a list keeping a training pointer is now as follows. (We will show the differences from the previous methods in red

-- you will see the the differences are usually quite minimal!)

```
        for(Node  p = head.next, q = head; p != null; q = p, p = p.next ) {
           // During first iteration, p points to first element and q to the header node;
           // thereafter, p points to a node and q points to the previous node
        }
```

For example, if we want to eliminate the first node in our list which contains a negative number, we would write the following:

```
       for(Node  p = head.next, q = head; p != null; q = p, p = p.next ) {
          if( p.item < 0 ) {
              q.next = p.next;
              break;
          }
```

You do not HAVE to use such a dummy header node, especially if you are not modifying the lists; when you are inserting or deleting nodes, however, they make your life simpler. In any case, they are unnecessary for recursive algorithms, as we shall see.

Iterative Algorithms for Linked Lists with Header Nodes

We now present the same algorithms in versions which assume a global head variable, and header node, as discussed above. They take the head pointer as a parameter, so that they can be used on multiple lists.

Print the List

```
    void printList(Node h)  {
        for(Node p = h.next ; p != null; p = p.next )
           System.out.println(p.item);
    }
```

You would call the method like this:

```
    printList(head);
```

Finding the length of a list

```
  int length(Node h) {
    int count = 0;
    for(Node p = h.next ; p!=null; p = p.next) {
      ++count;
    }
    return count;
  }
```

You would call the method like this:

```
    int n = length(head);
```

Looking up a item

```
    Node lookup(Node h,  int n) {
       for(Node p = h.next; p != null; p = p.next )
          if( p.item == n )
              return p;
       return null;             // return null if item not found
    }
```

You would call the method like this:

```
    Node q = lookup(head, 34);
```

Deleting an item in a list

Now we will see the big advantage of the header node technique: the header node removes cases 1 and 2, and only case 3 remains! The resulting algorithm is much simpler. In addition, you can now delete all instances of a given node by simply removing one line from the code!

```
   void delete(Node h,  int n ) {
       for(Node p = h.next, q = h; p != null ; q = p, p = p.next )
          if( p.item == n ) {
              q.next = p.next;
              return;              // if you delete this line it will remove all instances of the number
          }
     }
```

You would call the method like this:

```
    delete(head,23);
```

Inserting an item into a sorted list

Again, the algorithm is simpler, since cases 1 and 2 are gone!

```
  void  insertInOrder( Node h, int n ) {
        Node p = h.next;                    // p points to first real node in list
        Node q = h;                         // q trails p
        while( p != null ) {
           if(p.item >= n) {                // found insertion point, between p and q       -- This is case 3
                q.next = new Node(n, p);
                return;
           }
           q = p;
           p = p.next;
        }
        q.next = new Node(n, null);         // Node must be added at end of list    --- This is case 4
  }
```

You would call the method like this:

```
    insertInOrder(head,23);
```

Reversing a list

```
    void reverse(Node h) {
        Node p = h.next,  q = h,  r;
        while ( p != null ) {
           r = q;          // r follows q
           q = p;          // q follows p
           p = p.next;    // p moves to next node
           q.next = r;    // link q to preceding node
        }
        head.next.next = null;
        head.next = q;
     }
```

Recursive Algorithms

Recursively defined algorithms are a central part of any advanced programming course and occur in almost every aspect of computer science. Although they are difficult to understand initially, after one gets the knack, they are easier to write, debug, and understand than their iterative counterparts. In many cases, the only realistic solution possible for a certain problem is recursive.

Let us examine the definition of the factorial function. We can define the factorial of a number *n*, notated *n!*, in two ways:

1. *n!* is the product of all the integers between 1 and *n*, inclusive;
2. if *n* = 1, then *n!* = 1, otherwise *n!* = *n* * (*n-1*)!

The first definition gives us an explicit way to calculate *n!* which involves iterating through all the numbers from 1 to *n* and keeping a running sum; it could be expressed in Java as follows:

```
int factorial( int num ) {
   int fact = 1;

   for (int i = 1; i <= num; ++i)
      fact = fact * i;

   return(fact);
}
```

The second definition of *n!* is, at first glance, nonsense, because we are defining something in terms of itself. Its like asking someone what the food at a Thai restaurant is like and he tells you, ``Well, it's kind of like food from Thailand.'' Or you look up ``penultimate'' in the dictionary and it says ``just after propenultimate;'' but when you look up ``propenultimate'' it's defined as ``just before penultimate.'' Actually our example is not exactly this paradoxical, because we are defining our object, if you look closely, in terms of a slightly different object. That is, *n!* is defined in terms of ( *n-1*)!, which has a smaller value before the ``!''. Also, the definition has a condition: when the value of *n* is small enough, i.e., 1, the factorial is just given explicitly as 1. Since the recursive part always defines the factorial in terms of the factorial of a *smaller number*, we must reach 1 eventually. This is the trick which allows us to define mathematical objects in this way. We must define a mathematical function explicitly for some values, and then we can define other values in terms of the function itself, *as long as the function will eventually reach one of the explicit values.* Let us look at the Java for this version of the function:

```
int factorial( int num ) {
   if ( num == 1 )
      return 1;
   else return num * factorial( num - 1 );
}
```

This function has the following standard features of any recursively defined procedure or function:

1. it has an **if** or a **switch** statement;
2. this **if** statement tests whether the function input is one of the *base cases*, i.e., one for which a value is returned explicitly;
3. if the base case is found, an action is performed or a value is returned which does *not* involved calling the function again;
4. if the base case is not found, the function *calls itself* on an argument which is closer to the base case than the original argument.

When this program runs, the computer has to keep calling this function on increasingly smaller values of *n* until *n* equals 1. For example, to find the value of *Factorial(4)*, the computer has to find out the value of *Factorial(3)*; to find this value, it has to know the value of *Factorial(2)*; to get this value, it has to know the value of *Factorial(1)*. But it knows the value of *Factorial(1)*, since we told it that this is 1. Now it can find the value of *Factorial(2)*, etc. all the way back to *Factorial(4)*. It is important to realize that the computation of the Java function for a given value has to wait until it gets though with all the function calls it makes, even when it calls itself. Thus there will be many different *invocations* for the same piece of code even though only one of these will actually be executing; the rest will be waiting for the function calls they made to finish. You should try tracing the factorial program above for, say, *Factorial(5)*, to get a feel for the way it works.

Let's look at another simple recursive algorithm similar to the factorial function:

```
int power( int num, int exponent ) {
   if ( exponent == 1 )
      return num;
   else return num * power( num, exponent - 1 );
}
```

Here we are determining the value of an integer *num* raised to a power *exponent*. We could have written this explicitly by just creating a **for** loop to multiply *num* by itself *exponent* number of times, i.e., 5^4 = 5 * 5 * 5 * 5. But the recursive algorithm says that, for example, 5^4\ is just 5 * (5^3), which is just 5 * (5 * (5^2)), which is just 5 * (5 * (5 * (5^1))), which is just 5 * 5 * 5 * 5. So they really do the same thing in different ways. Note again that the recursive call involves the function calling itself on arguments which get closer to the base case--if you keep subtracting 1 from *exponent* you will eventually reach 1. Try this algorithm on *Power(2, 5)*.

Another recursive algorithm we could write would be for calculating the Nth Fibonacci number. Recall that the Fibonacci numbers form a series in which the first two values are both 1, and each successive value is the sum of the previous two values:

```
  1 1 2 3 5 8 13 21 34 55 89 ....
```

Thus the third Fibonacci number is 2, the seventh is 13, and so on. Note how the definition is phrased: ``the first two values are both 1" (an explicit answer is given), ``and each successive value is the sum of the previous values"(the rest are defined in terms of previous values in the series). This is clearly translatable into a recursive algorithm with almost no effort:

```
int  fibonacci( int n ) {
   if ( n < 2 )
      return 1;
   else return fibonacci(n-2) + fibonacci(n-1);
}
```

This is obviously a Java version of the English definition above, but will it work? After all, it calls itself not once but twice! The base case assures us, however, that this *must* stop eventually, since we call the function on smaller values each time. It must reach *Fibonacci(1)* or *Fibonacci(2)* eventually. In fact, this is not a very efficient way to calculate the Fibonacci numbers, since we must cover the same ground twice to get each number. It does work, however, and is an exact translation of the English definition we started with. In other words, it is a more natural expression of the original problem than an iterative algorithm, because the *original definition is recursive.* Again, try this on some small values to convince yourself that it works.

A slightly more difficult algorithm which can be written recursively is Euclid's *Greatest Common Divisor* algorithm, which has pride of place as the oldest recursive algorithm in existence. This ancient Greek mathematician discovered that we can find the largest integer which divides two given integers evenly if we generate a series of values a follows:

1. write down the two integers;
2. divide the first by the second and write down the *remainder* from that division;
3. if the remainder is 0, then the greatest common divisor is the number immediately to the left of the 0, i.e., the number you divided into the previous number to get a remainder of 0;
4. if the remainder is not 0, then repeat from step 2 using the last two integers in the list.

For example, starting with the two integers 28 and 18, we would generate the series:

```
  28  18  10  8  2  0.
```

Thus 2 is the greatest common divisor of 28 and 18. This method is essentially a recursive algorithm, although it may not be obvious at first. Notice that we perform the same action on each pair of numbers: we divide the first by the second and write down the remainder, then continue with the second number and the remainder just obtained, etc., until we reach 0. The recursive algorithm looks like this:

```
int gcd( int num1, int num2 ) {
   {
      if ( (num1 % num2) == 0 )
         return num2;
      else
```

```
        return gcd( num2, (num1 % num2) );
   }
```

This algorithm will implicitly create the list (except for the 0, which just indicates that the previous number divides the number before it evenly) that we showed above if you call *gcd(28, 18)*. It's tricky, but it does nothing really different than the recursive algorithms we have examined so far. It's worth tracing through on some simple input.

We have presented here a number of recursive functions returning values, but it is important to realize that void functions (not returning values) can be written recursively as well. For example, many sorting algorithms are written as recursively.

Other recursive algorithms are presented below for singly-linked lists and for tree structures. These are important algorithms which show the advantages of recursion clearly. For some, like the printing procedures for singly-linked lists, the iterative and recursive versions are of about equal complexity. For others, like the cell deletion algorithm, the difference is more pronounced in favor of the recursive version. For another class of algorithms, such as the tree walk and insertion procedures, the recursive version is really the only reasonable solution. In general, when a data structure is defined recursively(like a tree or a linked list) the most natural algorithms are recursive as well. To use these advanced data structures one must have a firm understanding of recursion. Those interested in pursuing this topic should try writing the recursive algorithms suggested in the notes on singly-linked lists below and should look up other recursive algorithms, such as Quicksort, Mergesort, the Tower of Hanoi, or practically any algorithm which involves trees.

Recursive Algorithms on Linked Lists

The recursive algorithms depend on a series of method calls to chain along the list, rather than an explicit for loop. The recursive versions of most linked-list algorithms are quite concise and elegant, compared with their iterative counterparts. For example, we do not need dummy header nodes; we will therefore not use them in this section.

Print the List

This is a simple algorithm, and good place to start. Recursion allows us flexibility in printing out a list forwards or in reverse (by exchanging the order of the recursive call):

```
void printList( Node p ) {
   if (p != null) {
       System.out.println(p.item);
       printList( p.next );
   }
 }


 void  printReverseList( Node p ) {
   if (p != null) {
       printReverseList( p.next );
       System.out.println(p.item);
   }
 }


 // Example of use:

 printList( head );     // Note: No dummy header node!
```

Finding the length of a list

Another simple recursive function.

```
int length( Node p ) {                  // Example call:   int len = length(head.next);
   if (p == null)
      return 0;
   else
      return 1 + length( p.next );
 }
```

When changing the structure of a linked list by deleting to adding nodes, it is useful to think in terms of reconstructing the list. Suppose you wanted to trivially reconstruct a list by reassigning all the references. You could do this:

Reconstruct a list

```
 Node construct( Node p ) {
   if( p == null )
      return null;
   else {
      p.next = construct( p.next );
      return p;
   }
}


// Example of use:
head.next = construct( head );
```

Pretty silly, right? But if we use this as a basis for altering the list recursively, then it becomes a very useful paradigm. All you have to do is figure out when to interrupt the silliness and do something useful. Here is a simple example, still kind of silly:

```
 Node addOne( Node p ) {
   if( p == null )
      return null;
   else {
      p.next = addOne( p.next );
      ++p.item;
      return p;
   }
}

// Example of use:

head.next = AddOne( head );
```

This recursively traverses the list and adds one to every item in the list. Following are some definitely non-silly algorithms using the approach to traversing the list.

This one is extremely simple, useful and not at all silly. Instead of reconstructing the same list, reconstruct another list, thereby building a copy (compare with the complicated iterative version above!):

Copy a list

```
 Node copy( Node p ) {
   if( p == null )
      return null;
   else
      return new Node(p.item, copy( p.next ));
}


// Example of use
newList = copy( head );
```

I'll repeat again that when using this "reconstruct the list" paradigm, we do NOT need to use a dummy header node to avoid the special case of the first node in the list; this next algorithm shows the advantage:

Inserting an item into a sorted list

```
Node insertInOrder( int k, Node p ) {
   if( p == null || p.item >= k )
      return new Node( k, p );
   else {
      p.next = insertInOrder( k, p.next );
      return p;
   }
}


// Example of use:
head = insertInOrder( 7, head );
```

Deleting an item from a list

This algorithm deletes the first occurrence of an item from a list. A simple change enables this algorithm to delete all occurrence of the item, by continuing to chain down the list after the item has been found. We assume that the list is unordered; you can easily change this to stop after finding a item beyond the search item by changing the first if condition.

```
Node delete( int k, Node p ) {
   if( p == null )                    // if the list is ordered use:  ( p == null || p.item > k )
      return p;
   else if ( p.item == k )
      return p.next;               // if you want to delete all instances, use:  return deleteItem( k, p.next );
   else {
      p.next = delete( k, p.next );
      return p;
   }
}
```

Deleting the last element in the list

This is a rather messy process in the iterative case; the use of recursion makes it much simpler:

```
public static  Node deleteLast( Node p ) {
   System.out.println("    " + p.item);
   if( p == null || p.next == null )
      return null;
   else {
      p.next = deleteLast( p.next );
      return p;
```

```
      }
   }
```

Appending two lists

Appending two lists is a simple way of creating a single list from two. This function adds the second list to the end of the first list:

```
    Node append( Node p, Node q ) {
       if ( p == null)
          return q;
       else {
         p.next = append( p.next, q );
         return p;
       }
    }
```

Example of use

```
   head = append(head, anotherList);
```

Merging two lists

Here is a more complex function to combine two lists; it simply zips up two lists, taking a node from one, then from the other. The first list in the original call now points to the new list.

```
    Node zip( Node p, Node q ) {
       if ( p == null)
          return q;
       else if ( q == null)
          return p;
       else {
          p.next = zip( q, p.next );     // Note how we exchange p and q here
          return p;
       }
    }
```

Example of call:

```
     head = zip( head, anotherlist );
```

If head points to 3 4 7 and anotherlist points to 2 5 6 8, then at the end of this call to zip, head will point to 3 2 4 5 7 6 8.

Merging two sorted lists

Here is another more complex function to combine two lists; this one merges nodes from two sorted lists, preserving their order:

```
    Node merge( Node p, Node q ) {
       if ( p == null)
          return q;
       else if ( q == null)
          return p;
       else if (p.item < q.item) {
          p.next = merge( p.next, q );
          return p;
       }
       else {
          q.next = merge( p, q.next );
          return q;
       }
    }
```

Example of call:

```
     head = merge( head, anotherlist );
```

If head points to 3 4 7 and anotherlist points to 2 5 6 8, then at the end of this call to zip, head will point to 2 3 4 5 6 8.

Other linked-list algorithms to try.....

Some other recursive algorithms(in increasing order of difficulty) you might want to try writing along the lines of those above are:

- Summing all the elements in an integer list, or finding the largest element
- Checking if two lists are identical
- Unzip a list into two lists, so that zip(unzip(h)) returns the same list h