Linked List Problems

By Nick Parlante

Abstract

This document presents 18 linked list problems covering a wide range of difficulty. Most obviously, these problems are useful to learn and practice linked list skills. More importantly, these problems are an excellent area to develop your ability with complex pointer algorithms. Even though modern languages and tools have made linked lists pretty unimportant for day-to-day programming, the skills for complex pointer algorithms are always in style, and linked lists are the classic area to develop those skills.

The problems use the C language syntax, so they require a basic understanding of C and its pointer syntax. The emphasis is on the important concepts of pointer manipulation and linked list algorithms rather than the features of the C language. The intermediate and advanced problems require the use of pointers to pointers — so called "reference pointers". See the references below for help with those background topics.

For a few problems multiple solutions are presented, such as iteration vs. recursion, dummy node vs. local reference. The specific problems are, in rough order of difficulty: Count, GetNth, DeleteList, Pop, InsertNth, SortedInsert, InsertSort, Append, FrontBackSplit, RemoveDuplicates, MoveNode, AlternatingSplit, ShuffleMerge, SortedMerge, SortedIntersect, Reverse, and RecursiveReverse.

Contents

Related CS Education Library Documents

Here are some helpful review materials for the linked list problems...

- Linked List Basics (http://cslibrary.stanford.edu/103/)
  Explains all the basic issues and techniques for building linked lists.

- Pointers and Memory (http://cslibrary.stanford.edu/102/)
  Explains all about how pointers and memory work in C and other languages. Starts with the very basics, and extends through advanced topics such as reference parameters and heap management.

- Essential C (http://cslibrary.stanford.edu/101/)
  Explains all the basic features of the C programming language.

Why Linked Lists Are Great To Study

Linked lists hold a special place in the hearts of many programmers. Linked lists are great to study because...

- *Nice Domain* The linked list structure itself is simple. Many linked list operations such as "reverse a list" or "delete a list" are easy to describe and understand since they build on the simple purpose and structure of the linked list itself.

- *Complex Algorithm* Even though linked lists are simple, the algorithms that operate on them can be as complex and beautiful as you want (See problem #18). It's easy to find linked list algorithms that are meaningful, complex, and pointer intensive. Linked list algorithms are not complex in a good way or a bad way, but in a realistic way for pointer code.

- *Pointer Intensive* Linked list problems are really about pointers. The linked list structure itself is obviously pointer intensive. Furthermore, linked list algorithms often break and re-weave the pointers in a linked list as they go. Linked lists give you a deep understanding of pointers.

- *Visualization* Visualization is an important skill in programming and design. Ideally, a programmer can visualize the state of memory to help think through the solution. Even the most abstract languages such as Java and Perl have layered, reference based data structures that benefit from visualization. Linked lists have a natural visual structure for practicing this sort of thinking. It's easy to draw the state of a linked list and use that drawing to think through the code.

Not to appeal to your mercenary side, but for all of the above reasons, linked list problems are often used as interview and exam questions. They are short to state, and have complex, pointer intensive solutions. No one really cares if you can build linked lists. But they do want to see if you have programming agility for complex algorithms and pointer manipulation, and linked lists are a great area of such problems.

What About a Simplifying Variations?

The problems in this document use the "plain" linked list, even though the code to manage the plain linked list is not the simplest. There are simplifying variations on the plain linked list that avoid some complex code — using dummy nodes, using a functional style to avoid reference pointers. For many programs the simplifying approach is a fine idea, but that is not what this document is about. The goal of this document is to seek out complex pointer problems and use them for practice. For that reason, the lists are used in their plain form, and the code will have all its complexity.

How To Use This Document

Try not to use these problems passively. You should take the time to try to solve each problem. Even if you do not succeed, you will think through the right issues in the attempt, and looking at the given solution will make more sense. Use drawings to think about the problems and work through the solutions. Linked lists are well-suited for memory drawings, so these problems are an excellent opportunity to develop your visualization skill.

Edition

This is the first major release of this document —Jan 17, 1999. The author may be reached at nick.parlante@cs.stanford.edu. The CS Education Library may be reached at cslibrary@cs.stanford.edu.

Dedication

This document is distributed for the benefit and education of all. That someone seeking education should have the opportunity to find it. May you learn from it in the spirit in which it is given — to make efficiency and beauty in your designs, peace and fairness in your actions.

Section 1 — Linked List Review

This section is a quick review of the concepts used in these linked list problems. For more detailed coverage, see Link List Basics (http://cslibrary.stanford.edu/103/) where all of this material is explained in much more detail.

Linked List Ground Rules

All of the linked list code in this document uses the "classic" singly linked list structure: A single head pointer points to the first node in the list. Each node contains a single `.next` pointer to the next node. The `.next` pointer of the last node is NULL. The empty list is represented by a NULL head pointer. All of the nodes are allocated in the heap.

For a few of the problems, the solutions present the temporary "dummy node" variation (see below), but most of the code deals with linked lists in their plain form. In the text, brackets {} are used to describe lists — the list containing the numbers 1, 2, and 3 is written as {1, 2, 3}. The node type used is...

```
struct node {
   int data;
   struct node* next;
};
```

For clarity, no typedefs are used. All pointers to nodes are declared simply as `struct node*`. Pointers to pointers to nodes are declared as `struct node**`. In the text, such pointers to pointers are often called "reference pointers".

Basic Utility Functions

In a few places, the text assumes the existence of the following basic utility functions...

- `int Length(struct node* head);`
  Returns the number of nodes in the list.

- `struct node* BuildOneTwoThree();`
  Allocates and returns the list {1, 2, 3}. Used by some of the example code to build lists to work on.

- `void Push(struct node** headRef, int newData);`
  Given an `int` and a reference to the head pointer (i.e. a `struct node**` pointer to the head pointer), add a new node at the head of the list with the standard 3-step-link-in: create the new node, set its `.next` to point to the current head, and finally change the head to point to the new node. (If you are not sure of how this function works, the first few problems may be helpful warm-ups.)

Use of the Basic Utility Functions

This sample code demonstrates the basic utility functions being used. Their implementations are also given in the appendix at the end of the document.

```
void BasicsCaller() {
   struct node* head;
   int len;

   head = BuildOneTwoThree(); // Start with {1, 2, 3}

   Push(&head, 13);           // Push 13 on the front, yielding {13, 1, 2, 3}
                              // (The '&' is because head is passed
                              //  as a reference pointer.)

   Push(&(head->next), 42);   // Push 42 into the second position
                              // yielding {13, 42, 1, 2, 3}
                              // Demonstrates a use of '&' on
                              // the .next field of a node.
                              // (See technique #2 below.)

   len = Length(head);        // Computes that the length is 5.
}
```

If these basic functions do not make sense to you, you can (a) go see Linked List Basics (http://cslibrary.stanford.edu/103/) which explains the basics of linked lists in detail, or (b) do the first few problems, but avoid the intermediate and advanced ones.

Linked List Code Techniques

The following list presents the most common techniques you may want to use in solving the linked list problems. The first few are basic. The last few are only necessary for the more advanced problems.

1) Iterate Down a List

A very frequent technique in linked list code is to iterate a pointer over all the nodes in a list. Traditionally, this is written as a `while` loop. The head pointer is copied into a local variable `current` which then iterates down the list. Test for the end of the list with `current!=NULL`. Advance the pointer with `current=current->next`.

```
// Return the number of nodes in a list (while-loop version)
int Length(struct node* head) {
   int count = 0;
   struct node* current = head;

   while (current != NULL) {
      count++;
      current = current->next
   }

   return(count);
}
```

Alternately, some people prefer to write the loop as a `for` which makes the initialization, test, and pointer advance more centralized, and so harder to omit...

```
for (current = head; current != NULL; current = current->next) {
```

2) Changing a Pointer With A Reference Pointer

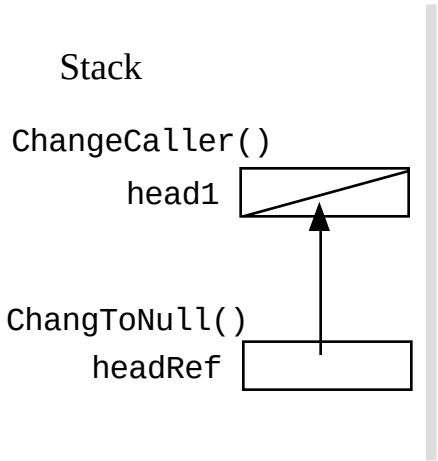
Many list functions need to change the caller's head pointer. To do this in the C language, pass a pointer to the head pointer. Such a pointer to a pointer is sometimes called a "reference pointer". The main steps for this technique are...

- Design the function to take a pointer to the head pointer. This is the standard technique in C — pass a pointer to the "value of interest" that needs to be changed. To change a `struct node*`, pass a `struct node**`.
- Use '&' in the caller to compute and pass a pointer to the value of interest.
- Use '*' on the parameter in the callee function to access and change the value of interest.

The following simple function sets a head pointer to NULL by using a reference parameter....

```
// Change the passed in head pointer to be NULL
// Uses a reference pointer to access the caller's memory
void ChangeToNull(struct node** headRef) {      // Takes a pointer to
                                                // the value of interest

   *headRef = NULL;     // use '*' to access the value of interest
}


void ChangeCaller() {
   struct node* head1;
   struct node* head2;

   ChangeToNull(&head1);      // use '&' to compute and pass a pointer to
   ChangeToNull(&head2);      // the value of interest
   // head1 and head2 are NULL at this point
}
```

Here is a drawing showing how the `headRef` pointer in ChangeToNull() points back to the variable in the caller...



Many of the functions in this document use reference pointer parameters. See the use of Push() above and its implementation in the appendix for another example of reference

pointers. See problem #8 and its solution for a complete example with drawings. For more detailed explanations, see the resources listed on page 1.

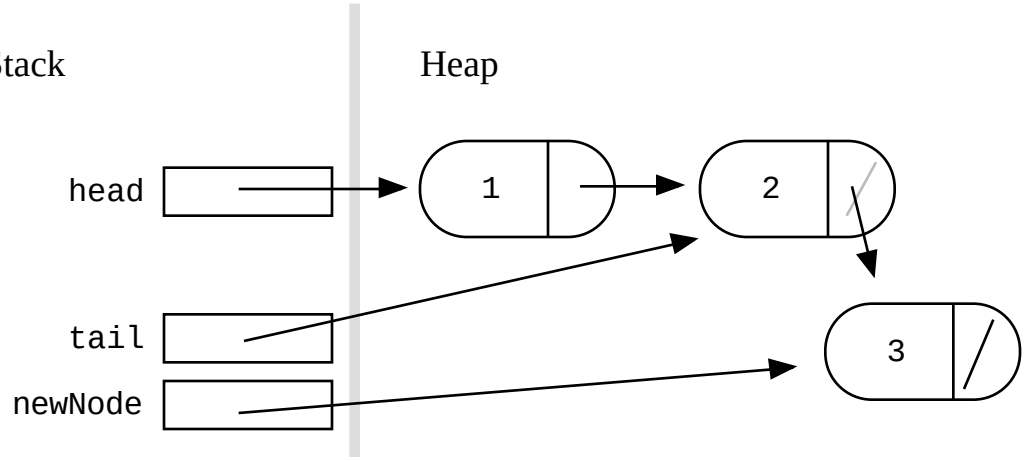
3) Build — At Head With Push()

The easiest way to build up a list is by adding nodes at its "head end" with Push(). The code is short and it runs fast — lists naturally support operations at their head end. The disadvantage is that the elements will appear in the list in the reverse order that they are added. If you don't care about order, then the head end is the best.

```
struct node* AddAtHead() {
   struct node* head = NULL;
   int i;

   for (i=1; i<6; i++) {
      Push(&head, i);
   }

   // head == {5, 4, 3, 2, 1};
   return(head);
}
```

4) Build — With Tail Pointer

What about adding nodes at the "tail end" of the list? Adding a node at the tail of a list most often involves locating the last node in the list, and then changing its `.next` field from NULL to point to the new node, such as the `tail` variable in the following example of adding a "3" node to the end of the list {1, 2}...



This is just a special case of the general rule: to insert or delete a node inside a list, you need a pointer to the node just *before* that position, so you can change its `.next` field. Many list problems include the sub-problem of advancing a pointer to the node before the point of insertion or deletion. The one exception is if the operation falls on the first node in the list — in that case the head pointer itself must be changed. The following examples show the various ways code can handle the single head case and all the interior cases...

5) Build — Special Case + Tail Pointer

Consider the problem of building up the list {1, 2, 3, 4, 5} by appending the nodes to the tail end. The difficulty is that the very first node must be added at the head pointer, but all the other nodes are inserted after the last node using a tail pointer. The simplest way to deal with both cases is to just have two separate cases in the code. Special case code first adds the head node {1}. Then there is a separate loop that uses a tail pointer to add all the

other nodes. The tail pointer is kept pointing at the last node, and each new node is added at `tail->next`. The only "problem" with this solution is that writing separate special case code for the first node is a little unsatisfying. Nonetheless, this approach is a solid one for production code — it is simple and runs fast.

```
struct node* BuildWithSpecialCase() {
   struct node* head = NULL;
   struct node* tail;
   int i;

   // Deal with the head node here, and set the tail pointer
   Push(&head, 1);
   tail = head;

   // Do all the other nodes using 'tail'
   for (i=2; i<6; i++) {
      Push(&(tail->next), i); // add node at tail->next
      tail = tail->next;      // advance tail to point to last node
   }

   return(head);  // head == {1, 2, 3, 4, 5};
}
```

6) Build — Dummy Node

Another solution is to use a temporary dummy node at the head of the list during the computation. The trick is that with the dummy, every node appear to be added after the `.next` field of a node. That way the code for the first node is the same as for the other nodes. The tail pointer plays the same role as in the previous example. The difference is that it now also handles the first node.

```
struct node* BuildWithDummyNode() {
   struct node dummy;   // Dummy node is temporarily the first node
   struct node* tail = &dummy; // Start the tail at the dummy.
                               // Build the list on dummy.next (aka tail->next)
   int i;

   for (i=1; i<6; i++) {
      Push(&(tail->next), i);
      tail = tail->next;
   }

   // The real result list is now in dummy.next
   // dummy.next == {1, 2, 3, 4, 5};
   return(dummy.next);
}
```

Some linked list implementations keep the dummy node as a permanent part of the list. For this "permanent dummy" strategy, the empty list is not represented by a NULL pointer. Instead, every list has a dummy node at its head. Algorithms skip over the dummy node for all operations. That way the dummy node is always present to provide the above sort of convenience in the code. Some of the solutions presented in this document will use the temporary dummy strategy. The code for the permanent dummy strategy is extremely similar, but is not shown.

7) Build — Local References

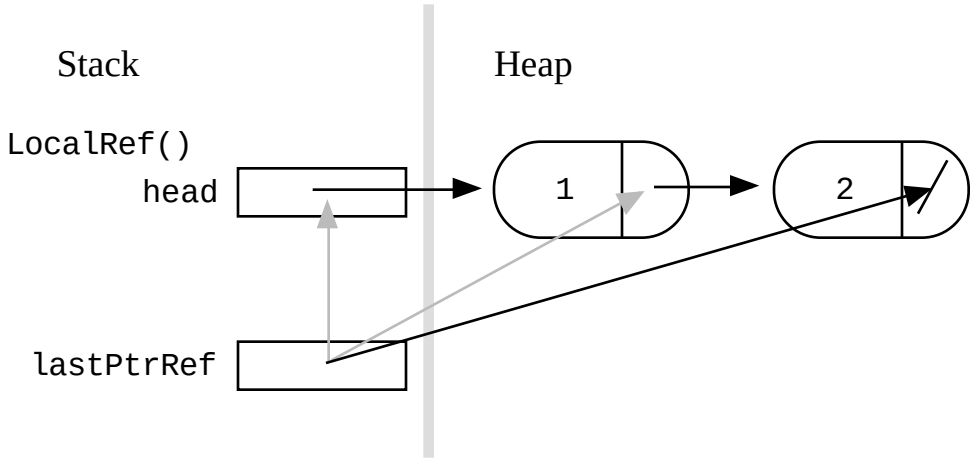
Finally, here is a tricky way to unifying all the node cases without using a dummy node. The trick is to use a local "reference pointer" which always points to the last *pointer* in the list instead of to the last node. All additions to the list are made by following the reference pointer. The reference pointer starts off pointing to the head pointer. Later, it points to the `.next` field *inside* the last node in the list. (A detailed explanation follows.)

```
struct node* BuildWithLocalRef() {
   struct node* head = NULL;
   struct node** lastPtrRef= &head; // Start out pointing to the head pointer
   int i;

   for (i=1; i<6; i++) {
      Push(lastPtrRef, i);    // Add node at the last pointer in the list
      lastPtrRef= &((*lastPtrRef)->next); // Advance to point to the
                                          // new last pointer
   }

   // head == {1, 2, 3, 4, 5};
   return(head);
}
```

This technique is short, but the inside of the loop is scary. This technique is rarely used. (Actually, I'm the only person I've known to promote it. I think it has a sort of compact charm.) Here's how it works...

1) At the top of the loop, `lastPtrRef` points to the last *pointer* in the list. Initially it points to the head pointer itself. Later it points to the `.next` field inside the last node in the list.
2) `Push(lastPtrRef, i);` adds a new node at the last pointer. The new node becomes the last node in the list.
3) `lastPtrRef= &((*lastPtrRef)->next);` Advance the `lastPtrRef` to now point to the `.next` field inside the new last node — that `.next` field is now the last pointer in the list.

Here is a drawing showing the state of memory for the above code just before the third node is added. The previous values of `lastPtrRef` are shown in gray...



This technique is never required to solve a linked list problem, but it will be one of the alternative solutions presented for some of the advanced problems.

Section 2 — Linked List Problems

Here are 18 linked list problems arranged by order of difficulty. The first few are quite basic and the last few are quite advanced. Each problem starts with a basic definition of what needs to be accomplished. Many of the problems also include hints or drawings to get you started. The solutions to all the problems are in the next section.

It's easy to just passively sweep your eyes over the solution — verifying its existence without lettings its details touch your brain. To get the most benefit from these problems, you need to make an effort to think them through. Whether or not you solve the problem, you will be thinking through the right issues, and the given solution will make more sense.

Great programmers can visualize data structures to see ahead to the solution, and linked lists are well suited to that sort of visual thinking. Use these problems to develop your visualization. skill. Make memory drawings to trace through the execution of code. Use drawings of the pre- and post-conditions of a problem to start thinking about a solution.

"The will to win means nothing without the will to prepare." - Juma Ikangaa, marathoner
(also attributed to Bobby Knight)

1 — Count()

Write a Count() function that counts the number of times an int occurs in a list. The code for this has the classic list traversal structure as demonstrated in Length().

```
void CountTest() {
   List myList = BuildOneTwoThree();      // build {1, 2, 3}

   int count = Count(myList, 2);    // returns 1 since there's 1 '2' in the list
}


/*
 Given a list and an int, return the number of times that int occurs
 in the list.
*/
int Count(struct node* head, int searchFor) {
// Your code
```

2 — GetNth()

Write a GetNth() function that takes a linked list and an integer index and returns the data value stored in the node at that index position. GetNth() uses the C numbering convention that the first node is index 0, the second is index 1, ... and so on. So for the list {42, 13, 666} GetNth() with index 1 should return 13. The index should be in the range [0..length-1]. If it is not, GetNth() should assert() fail (or you could implement some other error case strategy).

```
void GetNthTest() {
   List myList = BuildOneTwoThree();      // build {1, 2, 3}
   int lastNode = GetNth(myList, Length(myList)-1);   // returns the value 3
}
```

Essentially, GetNth() tries to make the list appear like an array — the client can ask for elements by index number. However, GetNth() is much slower than [ ] for an array. The advantage of the linked list is its much more flexible memory management — we can Push() at any time to add more elements and the memory is allocated as needed.

```
// Given a list and an index, return the data
// in the nth node of the list. The nodes are numbered from 0.
// Assert fails if the index is invalid (outside 0..lengh-1).
int GetNth(struct node* head, int index) {
// Your code
```

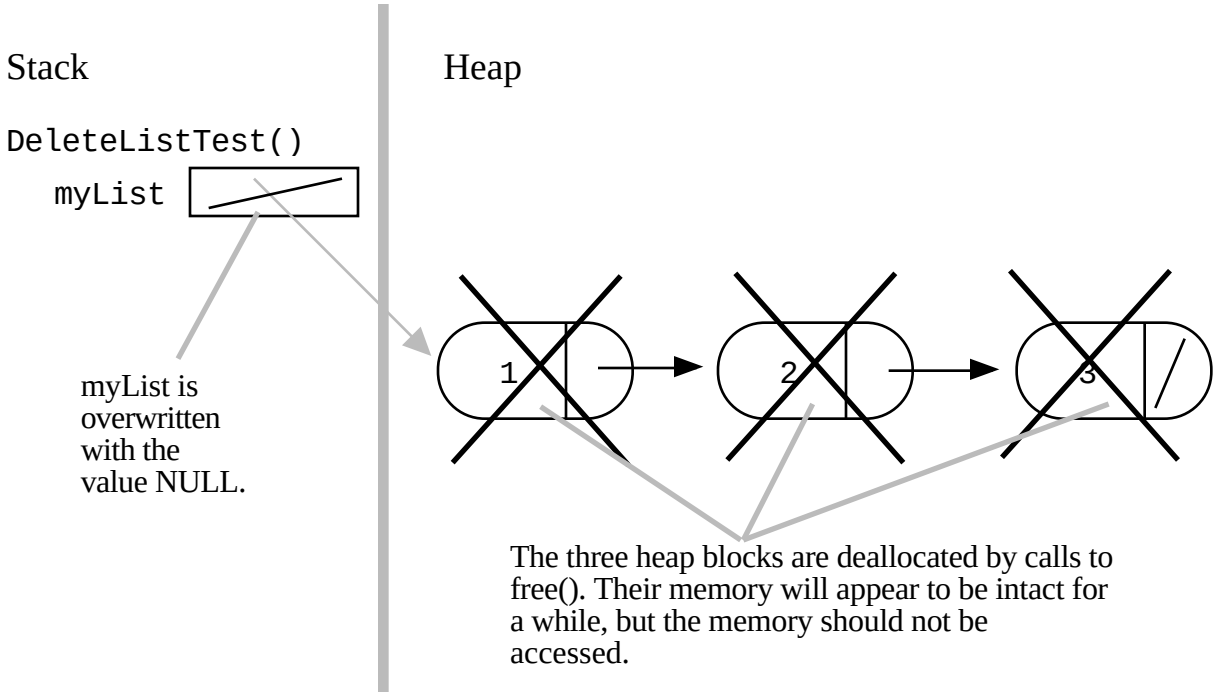
3 — DeleteList()

Write a function DeleteList() that takes a list, deallocates all of its memory and sets its head pointer to NULL (the empty list).

```
void DeleteListTest() {
   List myList = BuildOneTwoThree();      // build {1, 2, 3}

   DeleteList(&myList); // deletes the three nodes and sets myList to NULL
}
```

Post DeleteList() Memory Drawing

Here's the drawing showing the state of memory after DeleteList() executes in the above sample. Overwritten pointers are shown in gray and deallocated heap memory has an 'X' through it. Essentially DeleteList() just needs to call free() once for each node and set the head pointer to NULL.



DeleteList()

The DeleteList() implementation will need to use a reference parameter just like Push() so that it can change the caller's memory (myList in the above sample). The implementation also needs to be careful not to access the `.next` field in each node after the node has been deallocated.

```
void DeleteList(struct node** headRef) {
// Your code
```

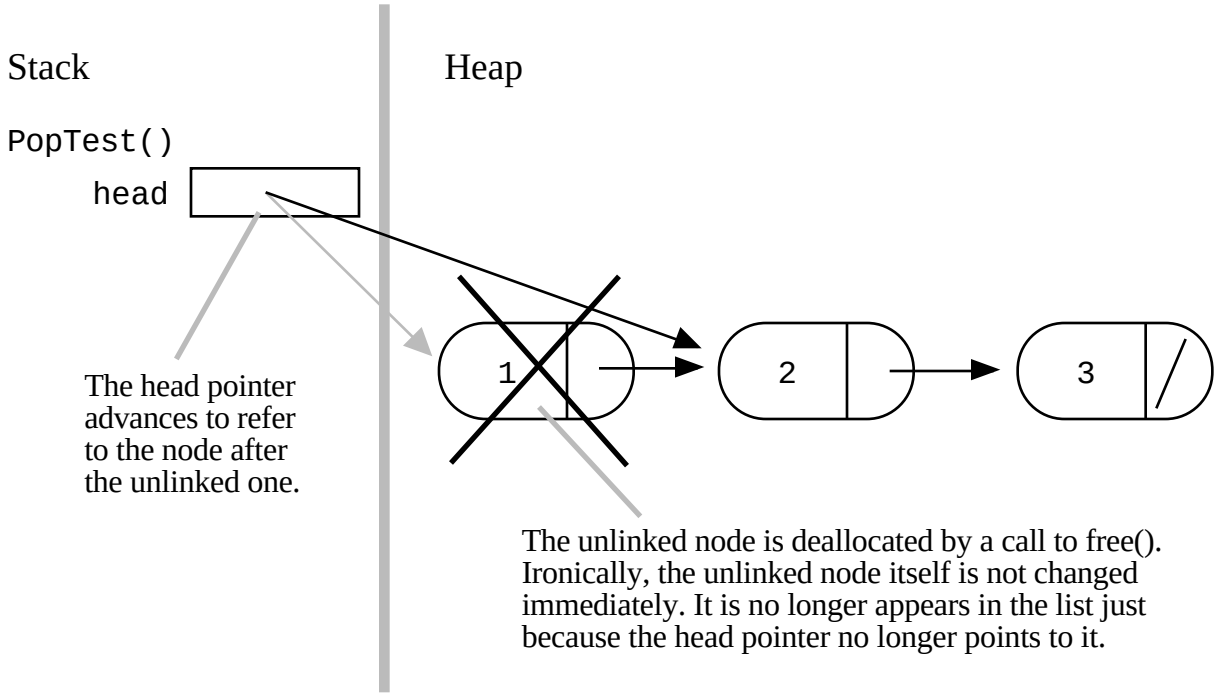
4 — Pop()

Write a Pop() function that is the inverse of Push(). Pop() takes a non-empty list, deletes the head node, and returns the head node's data. If all you ever used were Push() and Pop(), then our linked list would really look like a stack. However, we provide more general functions like GetNth() which what make our linked list more than just a stack. Pop() should assert() fail if there is not a node to pop. Here's some sample code which calls Pop()....

```
void PopTest() {
   struct node* head = BuildOneTwoThree();      // build {1, 2, 3}
   int a = Pop(&head);  // deletes "1" node and returns 1
   int b = Pop(&head);  // deletes "2" node and returns 2
   int c = Pop(&head);  // deletes "3" node and returns 3
   int len = Length(head);          // the list is now empty, so len == 0
}
```

Pop() Unlink

Pop() is a bit tricky. Pop() needs to unlink the front node from the list and deallocate it with a call to free(). Pop() needs to use a reference parameter like Push() so that it can change the caller's head pointer. A good first step to writing Pop() properly is making the memory drawing for what Pop() should do. Below is a drawing showing a Pop() of the first node of a list. The process is basically the reverse of the 3-Step-Link-In used by Push() (would that be "Ni Knil Pets-3"?). The overwritten pointer value is shown in gray, and the deallocated heap memory has a big 'X' drawn on it...



Pop()

```
/*
 The opposite of Push(). Takes a non-empty list
 and removes the front node, and returns the data
 which was in that node.
*/
int Pop(struct node** headRef) {
// your code...
```

5 — InsertNth()

A more difficult problem is to write a function InsertNth() which can insert a new node at any index within a list. Push() is similar, but can only insert a node at the head end of the list (index 0). The caller may specify any index in the range [0..length], and the new node should be inserted so as to be at that index.

```
void InsertNthTest() {
   struct node* head = NULL;  // start with the empty list

   InsertNth(&head, 0, 13);   // build {13)
   InsertNth(&head, 1, 42);   // build {13, 42}
   InsertNth(&head, 1, 5);          // build {13, 5, 42}

   DeleteList(&head);   // clean up after ourselves
}
```

InsertNth() is complex — you will want to make some drawings to think about your solution and afterwards, to check its correctness.

```
/*
 A more general version of Push().
 Given a list, an index 'n' in the range 0..length,
 and a data element, add a new node to the list so
 that it has the given index.
*/
void InsertNth(struct node** headRef, int index, int data) {
// your code...
```

6 — SortedInsert()

Write a SortedInsert() function which given a list that is sorted in increasing order, and a single node, inserts the node into the correct sorted position in the list. While Push() allocates a new node to add to the list, SortedInsert() takes an existing node, and just rearranges pointers to insert it into the list. There are many possible solutions to this problem.

```
void SortedInsert(struct node** headRef, struct node* newNode) {
// Your code...
```

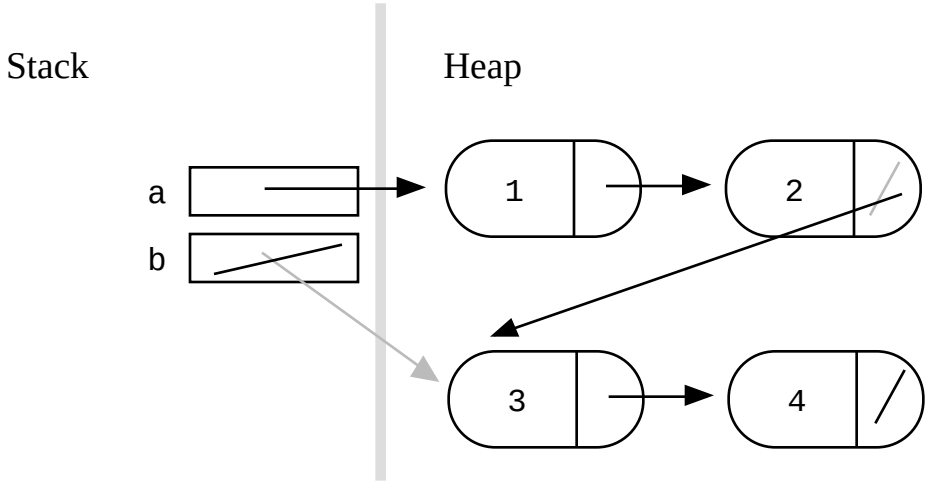
7 — InsertSort()

Write an InsertSort() function which given a list, rearranges its nodes so they are sorted in increasing order. It should use SortedInsert().

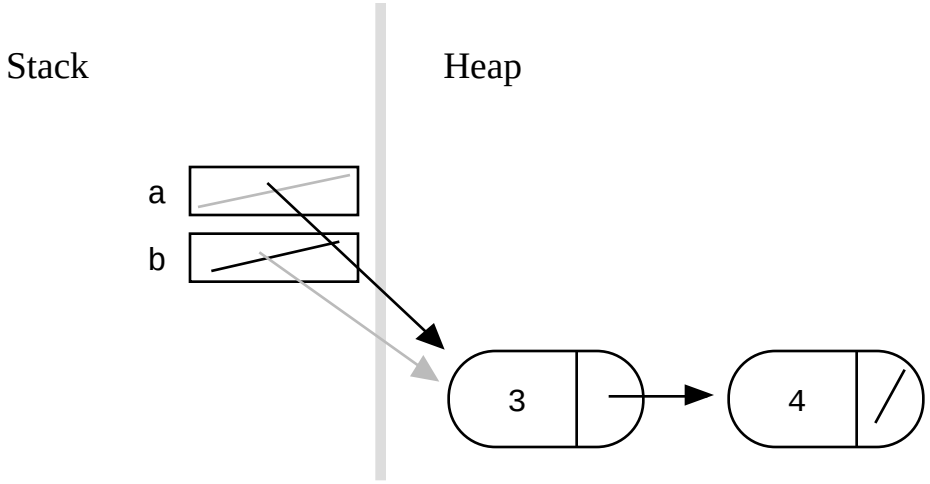
```
// Given a list, change it to be in sorted order (using SortedInsert()).
void InsertSort(struct node** headRef) { // Your code
```

8 — Append()

Write an Append() function that takes two lists, 'a' and 'b', appends 'b' onto the end of 'a', and then sets 'b' to NULL (since it is now trailing off the end of 'a'). Here is a drawing of a sample call to Append(a, b) with the start state in gray and the end state in black. At the end of the call, the 'a' list is {1, 2, 3, 4}, and 'b' list is empty.



It turns out that both of the head pointers passed to Append(a, b) need to be reference parameters since they both may need to be changed. The second 'b' parameter is always set to NULL. When is 'a' changed? That case occurs when the 'a' list starts out empty. In that case, the 'a' head must be changed from NULL to point to the 'b' list. Before the call 'b' is {3, 4}. After the call, 'a' is {3, 4}.



```
// Append 'b' onto the end of 'a', and then set 'b' to NULL.
void Append(struct node** aRef, struct node** bRef) {
// Your code...
```

9 — FrontBackSplit()

Given a list, split it into two sublists — one for the front half, and one for the back half. If the number of elements is odd, the extra element should go in the front list. So FrontBackSplit() on the list {2, 3, 5, 7, 11} should yield the two lists {2, 3, 5} and {7, 11}. Getting this right for all the cases is harder than it looks. You should check your solution against a few cases (length = 2, length = 3, length=4) to make sure that the list gets split correctly near the short-list boundary conditions. If it works right for length=4, it probably works right for length=1000. You will probably need special case code to deal with the (length <2) cases.

Hint. Probably the simplest strategy is to compute the length of the list, then use a `for` loop to hop over the right number of nodes to find the last node in the front half, and then cut the list at that point. A trickier way to do it is to have two pointers which traverse the list. One pointer advances two nodes at a time, while the other goes one node at a time. When the leading pointer reaches the end, the trailing pointer will be about half way. For either strategy, care is required to split the list at the right point.

```
/*
 Split the nodes of the given list into front and back halves,
 and return the two lists using the reference parameters.
 If the length is odd, the extra node should go in the front list.
*/
void FrontBackSplit(struct node* source,
                    struct node** frontRef, struct node** backRef) {
// Your code...
```

10 RemoveDuplicates()

Write a RemoveDuplicates() function which takes a list sorted in increasing order and deletes any duplicate nodes from the list. Ideally, the list should only be traversed once.

```
/*
 Remove duplicates from a sorted list.
*/
void RemoveDuplicates(struct node* head) {
// Your code...
```

11 — MoveNode()

This is a variant on Push(). Instead of creating a new node and pushing it onto the given list, MoveNode() takes two lists, removes the front node from the second list and pushes it onto the front of the first. This turns out to be a handy utility function to have for several later problems. Both Push() and MoveNode() are designed around the feature that list operations work most naturally at the head of the list. Here's a simple example of what MoveNode() should do...

```
void MoveNodeTest() {
   struct node* a = BuildOneTwoThree();   // the list {1, 2, 3}
   struct node* b = BuildOneTwoThree();

   MoveNode(&a, &b);
   // a == {1, 1, 2, 3}
   // b == {2, 3}
}

/*
 Take the node from the front of the source, and move it to
 the front of the dest.
 It is an error to call this with the source list empty.
*/
void MoveNode(struct node** destRef, struct node** sourceRef) {
// Your code
```

12 — AlternatingSplit()

Write a function AlternatingSplit() which takes one list and divides up its nodes to make two smaller lists. The sublists should be made from alternating elements in the original list. So if the original list is {a, b, a, b, a}, then one sublist should be {a, a, a} and the other should be {b, b}. You may want to use MoveNode() as a helper. The elements in the new lists may be in any order (for some implementations, it turns out to be convenient if they are in the reverse order from the original list.)

```
/*
 Give the source list, split its nodes into two shorter lists.
 aRef should be set to point to the list of odd position elements,
 and bRef should be set to point to the list of even position elements.
 The elements in the new lists may be in any order.
*/
void Split(struct node* source, struct node** aRef, struct node** bRef) {
// Your code
```

13— ShuffleMerge()

Given two lists, merge their nodes together to make one list, taking nodes alternately between the two lists. So ShuffleMerge() with {1, 2, 3} and {7, 13, 1} should yield {1, 7, 2, 13, 3, 1}. If either list runs out, the all the nodes should be taken from the other list. The solution depends on being able to move nodes to the end of a list as discussed in the Section 1 review. You may want to use MoveNode() as a helper. Overall, many techniques are possible: dummy node, local reference, or recursion. Using this function and FrontBackSplit(), you could simulate the shuffling of cards.

```
/*
 Merge the nodes of the two lists into a single list taking a nodes
 alternately from each list, and return the new list.
*/
struct node* ShuffleMerge(struct node* a, struct node* b) {
// Your code
```

14 — SortedMerge()

Write a SortedMerge() function which takes two lists, each of which is sorted in increasing order, and merges the two together into one list which is increasing order.

SortedMerge() should return the new list. The new list should be made by splicing together the nodes of the first two lists (use MoveNode()). Ideally, Merge() should only make one pass through each list. Merge() is tricky to get right — it may be solved iteratively or recursively. There are many cases to deal with: either 'a' or 'b' may be empty, during processing either 'a' or 'b' may run out first, and finally there's the problem of starting the result list empty, and building it up while going through 'a' and 'b'.

```
/*
 Takes two lists sorted in increasing order, and
 splices their nodes together to make one big
 sorted list which is returned.
*/
struct node* SortedMerge(struct node* a, struct node* b) {
// your code...
```

15 — MergeSort()

(This problem requires recursion) Given FrontBackSplit() and SortedMerge(), it's pretty easy to write a classic recursive MergeSort(): split the list into two smaller lists, recursively sort those lists, and finally merge the two sorted lists together into a single sorted list. Ironically, this problem is easier than either FrontBackSplit() or SortedMerge().

```
void MergeSort(struct node* headRef) {
// Your code...
```

16 — SortedIntersect()

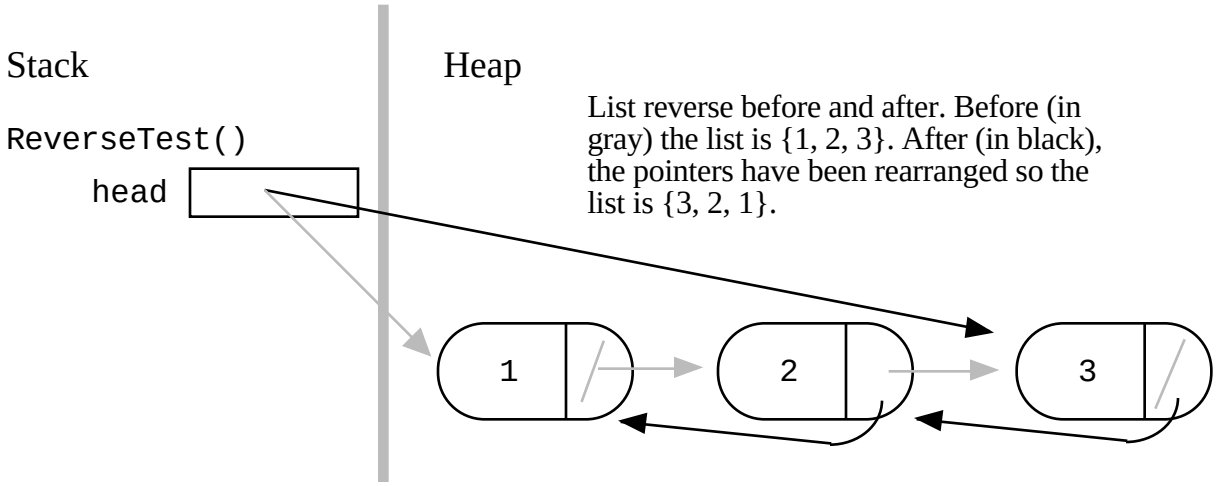
Given two lists sorted in increasing order, create and return a new list representing the intersection of the two lists. The new list should be made with its own memory — the original lists should not be changed. In other words, this should be Push() list building, not MoveNode(). Ideally, each list should only be traversed once. This problem along with Union() and Difference() form a family of clever algorithms that exploit the constraint that the lists are sorted to find common nodes efficiently.

```
/*
 Compute a new sorted list that represents the intersection
 of the two given sorted lists.
*/
struct node* SortedIntersect(struct node* a, struct node* b) {
// Your code
```

17 — Reverse()

Write an iterative Reverse() function which reverses a list by rearranging all the `.next` pointers and the head pointer. Ideally, Reverse() should only need to make one pass of the list. This is a difficult problem. It requires a good understanding of pointer operations as well as the ability to think through a complex code design. Any programmer can benefit from working through the code even if they need look at the solution first. The problem is a good example of the sort of code and memory issues which complex algorithms sometimes require. Many great programmers have struggled with Reverse() — it's a classic. (A memory drawing for Reverse() an some hints are below.)

```
void ReverseTest() {
   struct node* head;

   head = BuildOneTwoThree();
   Reverse(&head);
   // head now points to the list {3, 2, 1}

   DeleteList(&head);   // clean up after ourselves
}
```



Reverse Hint
As a hint, here's one strategy: Instead of running a single "current" pointer down the list, run three pointers (front, middle, back) down the list in order: front is on one node, middle is just behind it, and back in turn is one behind middle. Once the code to run the three pointers down the list is clear and tested in a drawing, add code to reverse the `.next` pointer of the middle node during the iteration. Add code to take care of the empty list and to adjust the head pointer itself.

```
/*
 Reverse the given linked list by changing its .next pointers and
 its head pointer. Takes a pointer (reference) to the head pointer.
*/
void Reverse(struct node** headRef) {
// your code...
```

18 — RecursiveReverse()

(This problem is difficult and is only possible if you are familiar with recursion.) The solution is short and amazing. As before, the code should only make a single pass over the list. Solving it, or even appreciating how it works requires real understanding of pointer code and recursion.

(Although the code for RecursiveReverse() is quite nice, there is another complex linked list recursion algorithm which is the most beautfiul of all — the O(n) conversion of an ordered binary tree to a sorted doubly linked circular list. That algorithm may get its own CS Education Library document someday.)

```
/*
 Recursively reverses the given linked list by changing its .next
 pointers and its head pointer. Takes a pointer (reference) to the
 head pointer.
*/
void RecursiveReverse(struct node** headRef) {
// your code...
```

Section 3 — Solutions

1 — Count() Solution

A straightforward iteration — just like Length().

```
int Count(struct node* head, int searchFor) {
   struct node* current = head;
   int count = 0;

   while (current != NULL) {
      if (current->data == searchFor) count++;
      current = current->next;
   }

   return count;
}
```

Alternately, the iteration may be coded with a `for` loop instead of a `while`...

```
int Count2(struct node* head, int searchFor) {
   struct node* current;
   int count = 0;

   for (current = head;   current != NULL;   current = current->next) {
      if (current->data == searchFor) count++;
   }

   return count;
}
```

2 — GetNth() Solution

Combine standard list iteration with the additional problem of counting over to find the right node. Off-by-one errors are common in this sort of code. Check it carefully against a simple case. If it's right for n=0, n=1, and n=2, it will probably be right for n=1000.

```
int GetNth(struct node* head, int index) {
   struct node* current = head;
   int count = 0; // the index of the node we're currently looking at

   while (current != NULL) {
      if (count == index) return(current->data);
      count++;
      current = current->next;
   }

   assert(0);  // if we get to this line, the caller was asking
               // for a non-existent element so we assert fail.
}
```

3 — DeleteList() Solution

Delete the whole list and set the head pointer to NULL. Slight complication in the inside of the loop, since we need extract the `.next` pointer before we delete the node, since after the delete it will be technically unavailable.

```
void DeleteList(struct node** headRef) {
   struct node* current = *headRef; // deref headRef to get the real head
   struct node* next;

   while (current != NULL) {
      next = current->next;   // note the next pointer
      free(current);          // delete the node
      current = next;         // advance to the next node
   }

   *headRef = NULL;     // Again, deref headRef to affect the real head back
                        // in the caller.
}
```

4 — Pop() Solution

Extract the data from the head node, delete the node, advance the head pointer to point at the next node in line. Uses a reference parameter since it changes the head pointer.

```
int Pop(struct node** headRef) {
   struct node* head;
   int result;

   head = *headRef;
   assert(head != NULL);

   result = head->data; // pull out the data before the node is deleted

   *headRef = head->next;     // unlink the head node for the caller (
                        // Note the * -- uses a reference-pointer
                        //  just like Push() and DeleteList().

   free(head);    // free the head node

   return(result);      // don't forget to return the data from the link
}
```

5 — InsertNth() Solution

This code handles inserting at the very front as a special case. Otherwise, it works by running a current pointer to the node before where the new node should go. Uses a for loop to march the pointer forward. The exact bounds of the loop (the use of < vs <=, n vs. n-1) are always tricky — the best approach is to get he general structure of the iteration correct first, and then make a careful drawing of a couple test cases to adjust the n vs. n-1 cases to be correct. (The so called "OBOB" — Off By One Boundary cases.) The OBOB cases are always tricky and not that interesting. Write the correct basic structure and then use a test case to get the OBOB cases correct. Once the insertion point has been determined, this solution uses Push() to do the link in. Alternately, the 3-Step Link In code could be pasted here directly.

```
void InsertNth(struct node** headRef, int index, int data) {
   // position 0 is a special case...
   if (index == 0) Push(headRef, data);
   else {
      struct node* current = *headRef;
      int i;

      for (i=0; i<index-1; i++) {
         assert(current != NULL);   // if this fails, index was too big
         current = current->next;
      }

      Push(&(current->next), data); // Tricky use of Push() --
                        // The pointer being pushed on is not
                        // in the stack. But actually this works
                        // fine -- Push() works for any node pointer.
   }
}
```

6 — SortedInsert() Solution

The basic strategy is to iterate down the list looking for the place to insert the new node. That could be the end of the list, or a point just before a node which is larger than the new node. The three solutions presented handle the "head end" case in different ways...

```
// Uses special case code for the head end
void SortedInsert(struct node** headRef, struct node* newNode) {
   // Special case for the head end
   if ((*headRef == NULL || (*headRef)->data>=newNode->data)) {
      newNode->next = *headRef;
      *headRef = newNode;
   }
   else {
      // Locate the node before the point of insertion
      struct node* current = *headRef;
      while (current->next!=NULL && current->next->data<newNode->data) {
         current = current->next;
      }
      newNode->next = current->next;
      current->next = newNode;
   }
}


// Dummy node strategy for the head end
void SortedInsert2(struct node** headRef, struct node* newNode) {
   struct node dummy;
   struct node* current = &dummy;
   dummy.next = *headRef;

   while (current->next!=NULL && current->next->data<newNode->data) {
      current = current->next;
   }

   newNode->next = current->next;
   current->next = newNode;

   *headRef = dummy.next;
}
```

```
// Local references strategy for the head end
void SortedInsert3(struct node** headRef, struct node* newNode) {
   struct node** currentRef = headRef;

   while (*currentRef!=NULL && (*currentRef)->data<newNode->data) {
      currentRef = &((*currentRef)->next);
   }

   newNode->next = (*currentRef)->next;
   *currentRef = newNode;
}
```

7 — InsertSort() Solution

Start with an empty result list. Iterate through the source list and SortedInsert() each of its nodes into the result list. Be careful to note the `.next` field in each node before moving it into the result list.

```
// Given a list, change it to be in sorted order (using InsertSorted()).
void InsertSort(struct node** headRef) {
   struct node* result = NULL;      // build the answer here
   struct node* current = *headRef; // iterate over the original list
   struct node* next;

   while (current!=NULL) {
      next = current->next;   // tricky - note the next pointer before we change it
      SortedInsert(&result, current);
      current = next;
   }

   *headRef = result;
}
```

8 — Append() Solution

The case where the 'a' list is empty is a special case handled first — in that case the 'a' head pointer needs to be changed directly. Otherwise we iterate down the 'a' list until we find its last node with the test `(current->next != NULL)`, and then tack on the 'b' list there. Finally, the original 'b' head is set to NULL. This code demonstrates extensive use of pointer reference parameters, and the common problem of needing to locate the last node in a list. (There is also a drawing of how Append() uses memory below.)

```
void Append(struct node** aRef, struct node** bRef) {
   struct node* current;

   if (*aRef == NULL) {       // Special case if a is empty
      *aRef = *bRef;
   }
   else {      // Otherwise, find the end of a, and append b there
      current = *aRef;
      while (current->next != NULL) {     // find the last node
         current = current->next;
      }
      current->next = *bRef;  // hang the b list off the last node
   }
```

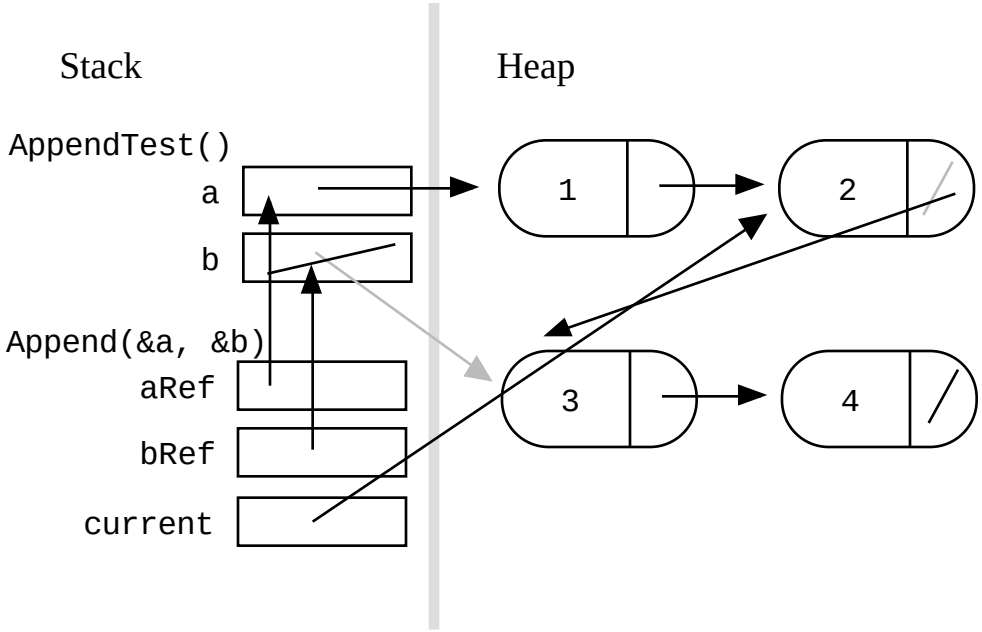
```
   *bRef=NULL;    // NULL the original b, since it has been appended above
}
```

Append() Test and Drawing

The following AppendTest() code calls Append() to join two lists. What does memory look like just before the call to Append() exits?

```
void AppendTest() {
   struct node* a;
   struct node* b;

   // set a to {1, 2}
   // set b to {3, 4}

   Append(&a, &b);
}
```

As an example of how reference parameters work, note how reference parameters in Append() point back to the head pointers in AppendTest()...



9 — FrontBachSplit() Solution

Two solutions are presented...

```
// Uses the "count the nodes" strategy
void FrontBackSplit(struct node* source,
                    struct node** frontRef, struct node** backRef) {

   int len = Length(source);
   int i;
   struct node* current = source;

   if (len < 2) {
      *frontRef = source;
```

```
      *backRef = NULL;
   }
   else {
      int hopCount = ((len+1)/2)-1; //(figured these with a few drawings)
      for (i = 0; i<hopCount; i++) {
         current = current->next;
      }

      // Now cut at current
      *frontRef = source;
      *backRef = current->next;
      current->next = NULL;
   }
}


// Uses the fast/slow pointer strategy
void FrontBackSplit2(struct node* source,
                  struct node** frontRef, struct node** backRef) {
   struct node* fast;
   struct node* slow;

   if (source==NULL || source->next==NULL) {    // length < 2 cases
      *frontRef = source;
      *backRef = NULL;
   }
   else {
      slow = source;
      fast = source->next;

      // Advance 'fast' two nodes, and advance 'slow' one node
      while (fast != NULL) {
         fast = fast->next;
         if (fast != NULL) {
            slow = slow->next;
            fast = fast->next;
         }
      }

      // 'Slow' is before the midpoint in the list, so split it in two
      // at that point.
      *frontRef = source;
      *backRef = slow->next;
      slow->next = NULL;
   }
}
```

10 — RemoveDuplicates() Solution

Since the list is sorted, we can proceed down the list and compare adjacent nodes. When adjacent nodes are the same, remove the second one. There's a tricky case where the node after the next node needs to be noted before the deletion.

```
// Remove duplicates from a sorted list
void RemoveDuplicates(struct node* head) {
   struct node* current = head;

   if (current == NULL) return;     // do nothing if the list is empty
```

```
   // Compare current node with next node
   while(current->next!=NULL) {
      if (current->data == current->next->data) {
         struct node* nextNext = current->next->next;
         free(current->next);
         current->next = nextNext;
      }
      else {
         current = current->next;   // only advance if no deletion
      }
   }
}
```

11 — MoveNode() Solution

The MoveNode() code is most similar to the code for Push(). It's short — just changing a couple pointers — but it's complex. Make a drawing.

```
void MoveNode(struct node** destRef, struct node** sourceRef) {
   struct node* newNode = *sourceRef;     // the front source node
   assert(newNode != NULL);

   *sourceRef = newNode->next;      // Advance the source pointer

   newNode->next = *destRef;  // Link the old dest off the new node
   *destRef = newNode;        // Move dest to point to the new node
}
```

12 — AlternatingSplit() Solution

The simplest approach iterates over the source list and use MoveNode() to pull nodes off the source and alternately put them on 'a' and b'. The only strange part is that the nodes will be in the reverse order that they occurred in the source list.

AlternatingSplit()

```
void AlternatingSplit(struct node* source,
                      struct node** aRef, struct node** bRef) {
   struct node* a = NULL;     // Split the nodes to these 'a' and 'b' lists
   struct node* b = NULL;

   struct node* current = source;
   while (current != NULL) {
      MoveNode(&a, &current); // Move a node to 'a'
      if (current != NULL) {
         MoveNode(&b, &current);    // Move a node to 'b'
      }
   }
   *aRef = a;
   *bRef = b;
}
```

AlternatingSplit() Using Dummy Nodes

Here is an alternative approach which builds the sub-lists in the same order as the source list. The trick here is to use a temporary dummy header nodes for the 'a' and 'b' lists as they are being built. Each sublist has a "tail" pointer which points to its current last node — that way new nodes can be appended to the end of each list easily. The dummy nodes

give the tail pointers something to point to initially. The dummy nodes are efficient in this case because they are temporary and allocated in the stack. Alternately, the "local references" technique could be used to get rid of the dummy nodes (see Section 1 for more details).

```
void AlternatingSplit2(struct node* source,
                       struct node** aRef, struct node** bRef) {
   struct node aDummy;
   struct node* aTail = &aDummy;    // points to the last node in 'a'
   struct node bDummy;
   struct node* bTail = &bDummy;    // points to the last node in 'b'
   struct node* current = source;

   aDummy.next = NULL;
   bDummy.next = NULL;

   while (current != NULL) {
      MoveNode(&(aTail->next), &current); // add at 'a' tail
      aTail = aTail->next;               // advance the 'a' tail
      if (current != NULL) {
         MoveNode(&(bTail->next), &current);
         bTail = bTail->next;
      }
   }

   *aRef = aDummy.next;
   *bRef = bDummy.next;
}
```

13 SuffleMerge() Solution

There are four separate solutions included. See Section 1 for information on the various dummy node and reference techniques.

SuffleMerge() — Dummy Node Not Using MoveNode()

```
struct node* ShuffleMerge(struct node* a, struct node* b) {
   struct node dummy;
   struct node* tail = &dummy;
   dummy.next = NULL;

   while (1) {
      if (a==NULL) {          // empty list cases
         tail->next = b;
         break;
      }
      else if (b==NULL) {
         tail->next = a;
         break;
      }
      else {                  // common case: move two nodes to tail
         tail->next = a;
         tail = a;
         a = a->next;

         tail->next = b;
         tail = b;
         b = b->next;
      }
   }
```

```
   return(dummy.next);
}
```

SuffleMerge() — Dummy Node Using MoveNode()

Basically the same as above, but use MoveNode().

```
struct node* ShuffleMerge(struct node* a, struct node* b) {
   struct node dummy;
   struct node* tail = &dummy;
   dummy.next = NULL;

   while (1) {
      if (a==NULL) {
         tail->next = b;
         break;
      }
      else if (b==NULL) {
         tail->next = a;
         break;
      }
      else {
         MoveNode(&(tail->next), &a);
         tail = tail->next;
         MoveNode(&(tail->next), &b);
         tail = tail->next;
      }
   }

   return(dummy.next);
}
```

SuffleMerge() — Local References

Uses a local reference to get rid of the dummy nodes entirely.

```
struct node* ShuffleMerge(struct node* a, struct node* b) {
   struct node* result = NULL;
   struct node** lastPtrRef = &result;

   while (1) {
      if (a==NULL) {
         *lastPtrRef = b;
         break;
      }
      else if (b==NULL) {
         *lastPtrRef = a;
         break;
      }
      else {
         MoveNode(lastPtrRef, &a);
         lastPtrRef = &((*lastPtrRef)->next);
         MoveNode(lastPtrRef, &b);
         lastPtrRef = &((*lastPtrRef)->next);
      }
   }

   return(result);
}
```

SuffleMerge() — Recursive

The recursive solution is the most compact of all, but is probably not appropriate for production code since it uses stack space proportionate the lengths of the lists.

```
struct node* ShuffleMerge(struct node* a, struct node* b) {
   struct node* result;
   struct node* recur;

   if (a==NULL) return(b);          // see if either list is empty
   else if (b==NULL) return(a);
   else {
      // it turns out to be convenient to do the recursive call first --
      // otherwise a->next and b->next need temporary storage.

      recur = ShuffleMerge(a->next, b->next);

      result = a;       // one node from a
      a->next = b;      // one from b
      b->next = recur;  // then the rest
      return(result);
   }
}
```

14 — SortedMerge() Solution

SortedMerge() Using Dummy Nodes

The strategy used here is to use a single "dummy" node as the start of the result list. The pointer `tail` always points to the last node in the result list, so appending new nodes is easy. The dummy node gives `tail` something to point to initially when the result list is empty. This dummy node is efficient, since it is only temporary, and it is allocated in the stack. The loop proceeds, removing one node from either 'a' or 'b', and adding it to `tail`. When we are done, the result is in dummy.next.

```
struct node* SortedMerge(struct node* a, struct node* b) {
   struct node dummy;   // a dummy first node to hang the result on
   struct node* tail = &dummy;      // Points to the last result node --
                                    // so tail->next is the place to add
                                    // new nodes to the result.

   dummy.next = NULL;

   while (1) {
      if (a == NULL) {  // if either list runs out, use the other list
         tail->next = b;
         break;
      }
      else if (b == NULL) {
         tail->next = a;
         break;
      }

      if (a->data <= b->data) {
         MoveNode(&(tail->next), &a);
      }
      else {
         MoveNode(&(tail->next), &b);
```

```
      }
      tail = tail->next;
   }

   return(dummy.next);
}
```

SortedMerge() Using Local References

This solution is structurally very similar to the above, but it avoids using a dummy node. Instead, it maintains a struct node** pointer, `lastPtrRef`, that always points to the last *pointer* of the result list. This solves the same case that the dummy node did — dealing with the result list when it is empty. If you are trying to build up a list at its tail, either they dummy node or the struct node** "reference" strategies can be used (see Section 1 for details).

```
struct node* SortedMerge2(struct node* a, struct node* b) {
   struct node* result = NULL;
   struct node** lastPtrRef = &result;    // point to the last result pointer

   while (1) {
      if (a==NULL) {
         *lastPtrRef = b;
         break;
      }
      else if (b==NULL) {
         *lastPtrRef = a;
         break;
      }

      if (a->data <= b->data) {
         MoveNode(lastPtrRef, &a);
      }
      else {
         MoveNode(lastPtrRef, &b);
      }
      lastPtrRef = &((*lastPtrRef)->next);      // tricky: advance to point to
                                                // the next ".next" field

   }

   return(result);
}
```

SortedMerge() Using Recursion

Merge() is one of those problems where the recursive solution has fewer weird cases than the iterative solution. You probably wouldn't want to use the recursive version for production code however, because it will use stack space which is proportional the length of the lists.

```
struct node* SortedMerge3(struct node* a, struct node* b) {
   struct node* result = NULL;

   // Base cases
   if (a==NULL) return(b);
   else if (b==NULL) return(a);

   // Pick either a or b, and recur
```

```
   if (a->data <= b->data) {
      result = a;
      result->next = SortedMerge3(a->next, b);
   }
   else {
      result = b;
      result->next = SortedMerge3(a, b->next);
   }

   return(result);
}
```

15 — MergeSort() Solution

The MergeSort strategy is: split into sublists, sort the sublists recursively, merge the two sorted lists together to form the answer.

```
void MergeSort(struct node** headRef) {
   struct node* head = *headRef;
   struct node* a;
   struct node* b;

   // Base case -- length 0 or 1
   if ((head == NULL) || (head->next == NULL)) {
      return;
   }

   FrontBackSplit(head, &a, &b);    // Split head into 'a' and 'b' sublists
                                    // We could just as well use AlternatingSplit()

   MergeSort(&a); // Recursively sort the sublists
   MergeSort(&b);

   *headRef  = SortedMerge(a, b);   // answer = merge the two sorted lists together
}
```

(Extra for experts) Using recursive stack space proportional to the length of a list is not recommended. However, the recursion in this case is ok — it uses stack space which is proportional to the *log* of the length of the list. For a 1000 node list, the recursion will only go about 10 deep. For a 2000 node list, it will go 11 deep. If you think about it, you can see that doubling the size of the list only increases the depth by 1.

16 — SortedIntersect()

The strategy is to advance up both lists and build the result list as we go. When the current point in both lists are the same, add a node to the result. Otherwise, advance whichever list is smaller. By exploiting the fact that both lists are sorted, we only traverse each list once. To build up the result list, both the dummy node and local reference strategy solutions are shown...

```
// This solution uses the temporary dummy to build up the result list
struct node* SortedIntersect(struct node* a, struct node* b) {
   struct node dummy;
   struct node* tail = &dummy;

   dummy.next = NULL;

   // Once one or the other list runs out -- we're done
```

```
   while (a!=NULL && b!=NULL) {
      if (a->data == b->data) {
         Push((&tail->next), a->data);
         tail = tail->next;
         a = a->next;
         b = b->next;
      }
      else if (a->data < b->data) { // advance the smaller list
         a = a->next;
      }
      else {
         b = b->next;
      }
   }

   return(dummy.next);
}


// This solution uses the local reference
struct node* SortedIntersect2(struct node* a, struct node* b) {
   struct node* result = NULL;
   struct node** lastPtrRef = &result;

   // Advance comparing the first nodes in both lists.
   // When one or the other list runs out, we're done.
   while (a!=NULL && b!=NULL) {
      if (a->data == b->data) {     // found a node for the intersection
         Push(lastPtrRef, a->data);
         lastPtrRef = &((*lastPtrRef)->next);
         a=a->next;
         b=b->next;
      }
      else if (a->data < b->data) { // advance the smaller list
         a=a->next;
      }
      else {
         b=b->next;
      }
   }

   return(result);
}
```

17 — Reverse() Solution

This uses the back-middle-front strategy described in the problem statement. We only make one pass of the list.

```
// Reverses the given linked list by changing its .next pointers and
// its head pointer. Takes a pointer (reference) to the head pointer.
void Reverse(struct node** headRef) {
   if (*headRef != NULL) {          // special case: skip the empty list

/*
 Plan for this loop: move three pointers: front, middle, back
 down the list in order. Middle is the main pointer running
 down the list. Front leads it and Back trails it.
 For each step, reverse the middle pointer and then advance all
```

```
 three to get the next node.
*/

      struct node* middle = *headRef;           // the main pointer

      struct node* front = middle->next;  // the two other pointers (NULL ok)
      struct node* back = NULL;


      while (1) {
         middle->next = back;       // fix the middle node

         if (front == NULL) break;  // test if done

         back = middle;             // advance the three pointers
         middle = front;
         front = front->next;
      }

      *headRef = middle;      // fix the head pointer to point to the new front
   }
}
```

18 — RecursiveReverse() Solution

Probably the hardest part is accepting the concept that the `RecursiveReverse(&rest)` does in fact reverse the rest. Then then there's a trick to getting the one front node all the way to the end of the list. Make a drwaing to see how the trick works.

```
void RecursiveReverse(struct node** headRef) {
   struct node* first;
   struct node* rest;

   if (*headRef == NULL) return;          // empty list base case

   first = *headRef;    // suppose first = {1, 2, 3}
   rest = first->next;  //         rest = {2, 3}

   if (rest == NULL) return;        // empty rest base case

   RecursiveReverse(&rest);   // Recursively reverse the smaller {2, 3} case
                              // after: rest = {3, 2}

   first->next->next = first; // put the first elem on the end of the list
   first->next = NULL;        // (tricky step -- make a drawing)

   *headRef = rest;     // fix the head pointer
}
```

Appendix

Basic Utility Function Implementations

Here is the source code for the basic utility functions.

Length()

```
// Return the number of nodes in a list
int Length(struct node* head) {
   int count = 0;
   struct node* current = head;

   while (current != NULL) {
      count++;
      current=current->next;
   }

   return(count);
}
```

Push()

```
// Given a reference (pointer to pointer) to the head
// of a list and an int, push a new node on the front of the list.
// Creates a new node with the int, links the list off the .next of the
// new node, and finally changes the head to point to the new node.
void Push(struct node** headRef, int newData) {
   struct node* newNode = malloc(sizeof(struct node));      // allocate node
   newNode->data = newData;                                 // put in the data
   newNode->next = (*headRef);      // link the old list off the new node
   (*headRef) = newNode;            // move the head to point to the new node
}
```

BuildOneTwoThree()

```
// Build and return the list {1, 2, 3}
struct node* BuildOneTwoThree() {
   struct node* head = NULL;  // Start with the empty list
   Push(&head, 3);      // Use Push() to add all the data
   Push(&head, 2);
   Push(&head, 1);

   return(head);
}
```