

```
// 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) {
        // for (current = head; current != NULL; current = current->next) {
        count++;
        current = current->next;
    }
    return(count);
}
```

```
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.
}
```

1 — Count() Solution

A straightforward iteration down the list — 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. There is a slight complication inside 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
}
```

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.)

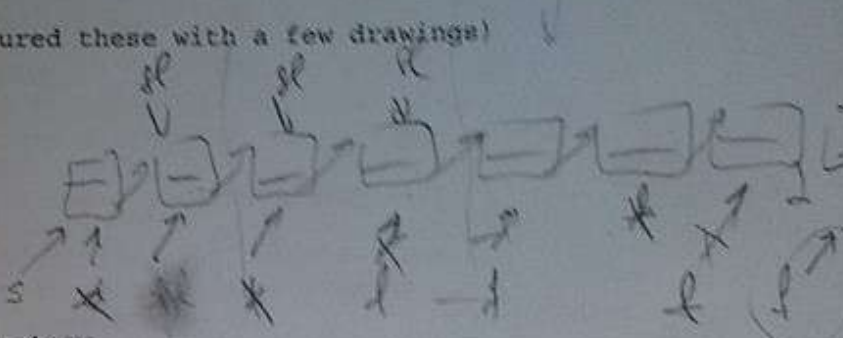
```
void Append(struct node** aRef, struct node** bRef) {
    struct node* current;
    if (*aRef == NULL) { // Special case if a is empty
        *aRef = *bRef;
    }
    else { // Otherwise, find the end of a, and append b there
        current = *aRef;
        while (current->next != NULL) { // find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


9 — FrontBackSplit() 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;
25
if (len < 2) {
*frontRef = source;
*backRef = NULL;
}
else {
int hopCount = (len-1)/2; //(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

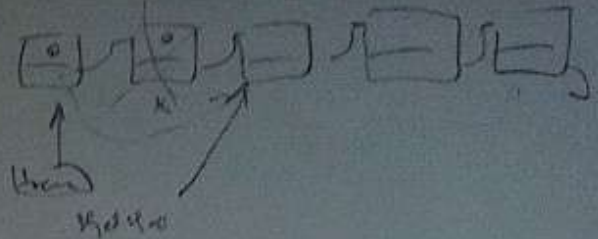
Since the list is sorted, we can proceed down the list and compare adjacent nodes. When adjacent nodes are the same, remove the second one. There's a tricky case where the node after the next node needs to be noted before the deletion.

```
// 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
    }
}

```



17 — Reverse() Solution

Finally, here's the back-middle-front strategy...

// 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
        /*

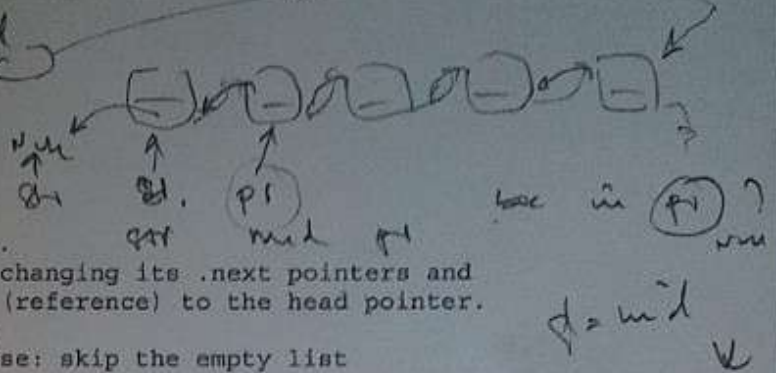
```

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 drawing 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
}

```

