Interview Question: Linked List Shuffling

Difficulty: Hard

Suppose you have a deck of cards represented as a linked list. You can ***perfectly shuffle*** that list by cutting it at the halfway point, then interleaving the two halves by alternating back and forth between the cards. For example, suppose you want to perfectly shuffle this sequence:

1 2 3 4 5 6 7 8 9 10

You'd start by splitting it into two halves, like this:

1 2 3 4 5 **6 7 8 9 10**

Then, you'd interleave the halves, like this:

6 1 7 2 8 3 9 4 10 5

The resulting list is said to have been perfectly shuffled. Your job is to write a function that accepts as input a pointer to a linked list with an even number of elements, then rearranges the elements in that list so that they're perfectly shuffled.

Possible Follow-Up Questions:

- Should we use a "raw" linked list, or a library type like LinkedList or std::list? **You should use "raw" linked lists, and you'll probably want to define your own node type.**
- Are we given, in advance, the number of elements in the list? **No**.
- Is the linked list singly-linked or doubly-linked? **Either is fine**.
- Can we assume the input list has an even number of elements? **Yes**.
- Do we have to worry about the empty list? **Yes**.
- Can we allocate any new linked list cells? **No**.
- Can we assume the linked list has a dummy head or tail? **If you'd like**.

What to Watch For

- Watch out to make sure that they don't allocate any new list cells. You shouldn't see "new" or "malloc" anywhere in their solution.
- Make sure that their code doesn't ever dereference a null pointer!
- Make sure that when they split the list in half they break the link from the first half to the second. If not, their code may accidentally introduce a cycle into the list.
- Their code needs to update the head pointer, either by returning the new head or by taking the pointer in by reference (or as a pointer to a pointer.)

Possible Solutions

Roughly speaking, the solution consists of two phases: first, split the list in half; second, shuffle the elements together. Splitting the list in half is harder than it looks – you need to figure out where the midpoint is and break the link connecting the first half to the second. There are a lot of different ways to do this. Here's a rough outline of what the code might look like:

```
void shuffleList(Node** head) {
      if (* head == NULL) return;

      Node* half = splitAtHalf(*head);
      interleave(*head, half);
      *head = half;
}
```

Finding the Halfway Point

One simple option is to count how many elements there are in the list, then scan to the halfway point, breaking the last link followed.

```
Node* splitAtHalf(Node* first) {
      size_t numElems = 0;
      for (Node* curr = first; curr != NULL; curr = curr->next) {
            numElems++;
      }

      /* Go one spot before the halfway point. */
      for (size_t i = 0; i < numElems / 2 - 1; i++) {
            first = first->next;
      }

      /* Split the list at this point. */
      Node* result = first->next;
      first->next = NULL;
      return result;
}
```

You may also see this solution, which works by sending two pointers down the list: one moving at speed one, and one moving at speed two. As soon as the one at speed two hits the end, you know that the one at speed one is at the halfway point.

```
Node* splitAtHalf(Node* first) {
      Node* fast = first;
      while (fast->next->next != NULL) {
            fast = fast->next->next;
            first = first->next;
      }

      /* Split the list at this point. */
      Node* result = first->next;
      first->next = NULL;
      return result;
```

```
}
```

The Interleave Step

One option is to interleave the lists iteratively. We'll keep a pointer to the last element of the list that we're building up so that we can keep appending elements to it. This ultimately returns the first element of the new, interleaved list.

The approach given below manually builds the list one element at a time.

```
void interleave(Node* first, Node* second) {
      Node* tail = NULL;
      while (second != NULL) {
            /* Append the first element of 'second' to the list. */
            if (tail == NULL) {
                  tail = second;
            } else {
                  tail->next = second;
            }

            /* This element now points to the first element of the first
             * list, which is now the tail. However, we also need to update
             * the 'second' list in the process so that we don't lose where
             * it is.
             */
            Node* next = second->next;
            second->next = first;
            second = next;
            tail = first;

            /* Remove the first element of 'first.' */
            next = first->next;
            first->next = NULL;
            first = next;
      }
}
```

Another approach to this is to iteratively tack the first element of one of the lists onto the back of the list being built, then to switch which list is which. This is shown here:

```
void interleave(Node* first, Node* second) {
      Node* tail = NULL;
      while (second != NULL) {
            /* Append the first element of 'second' to the list. */
            if (tail == NULL) {
                  tail = second;
            } else {
                  tail->next = second;
                  tail = second;
            }

            /* Cut the head of 'second' from 'second.' */
            Node* next = second->next;
            second->next = NULL;
            second = next;

            /* Swap the two lists. */
            Node* temp = first;
            first = second;
            second = first;
      }
}
```

You can also do this recursively, which might be even cleaner:

```
void interleave(Node* first, Node* second) {
      Node* tail = NULL;
      recInterleave(first, second, &tail);
}
Node* recInterleave(Node* first, Node* second, Node** tail) {
      if (second == NULL) return NULL;

      if (*tail == NULL) {
            *tail = second;
      } else {
            (*tail)->next = second;
            *tail = second;
      }

      /* The next pointer should be what you get by interleaving
       * the two lists, but with their roles reversed.
       */
      second->next = recInterleave(second->next, first, tail);

      /* The ultimate head of the list is the first element of 'second' */
      return second;
}
```