Section Solution

Discussion Problem 1 Solution: Braided Lists

The recursive approach is easier to code up, but requires more a much more clever algorithm. (In practice, it would be unacceptable though, since there's an almost-as-easy iterative option, and you almost always go with the iterative version over a unary recursion if at all possible. Are you going to allow 10,000 recursive calls to be made to braid a list of length 10,000? You really shouldn't).

```
static void braidRec(node *list, Queue<int>& numbers) {
   if (list == NULL) return;
   numbers.enqueue(list->value);
   braidRec(list->next, numbers);
   node *newNode = new node;
   newNode->value = numbers.dequeue();
   newNode->next = list->next;
   list->next = newNode;
}

static void braid(node *list) {
   Queue<int> numbers;
   braidRec(list, numbers);
}
```

A fully iterative approach is best handled in two passes:

```
static void braid(node *list) {
   node *reverse = NULL;
   for (node *curr = list; curr != NULL; curr = curr->next) {
      node *newNode = new node;
      newNode->value = curr->value;
      newNode->next = reverse;
      reverse = newNode;
   }

   // reverse now addresses a memory-independent version of the original list,
   // where all of the nodes are in reverse order.

   node *rest = reverse; // rest addresses part that has yet to be braided in
   for (node *curr = list; curr != NULL; curr = curr->next->next) {
      node *next = rest->next;
      rest->next = curr->next;
      curr->next = rest;
      rest = next;
   }
}
```

Note: for some reason, industry interviews tend to include a question that asks you to reverse a linked list. Commit the first half of the iterative solution to memory. ☺

Discussion Problem 2 Solution: Skip Lists

This is conceptually dense, but an optimal implementation is fairly short. There are several ways to implement this, and I take the approach of tracking the address of the relevant links vector at any one moment, and decide whether I can advance to the node with just as many forward links, or whether I need to descend down through the current links vector and be less aggressive about skipping forward.

```
static bool skipListContains(Vector<skipListNode *>& heads, int key) {
   Vector<skipListNode *> *currs = &heads;
   int level = currs->size() - 1;

   while (level >= 0) {
      skipListNode *curr = (*currs)[level];
      if (curr != NULL && curr->value == key) return true;
      if (curr == NULL || curr->value > key) {
         level--;
      } else {
         currs = &(curr->links);
      }
   }

   return false;
}
```

Discussion Problem 3 Solution: Ranked Choice Voting

a.

```
static string identifyLeastPopular(ballot *ballots) {
   Map<string, int> counts;
   for (ballot *curr = ballots; curr != NULL; curr = curr->next) {
      counts[curr->votes[0]]++;
   }

   int threshold = -1;
   string leastPopular;
   foreach (string candidate in counts) {
      if (threshold == -1 || counts[candidate] < threshold) {
         leastPopular = candidate;
         threshold = counts[candidate];
      }
   }

   return leastPopular;
}
```

The problem was written with the assumption that, at least initially, all candidates ever mentioned have at least one first-choice vote. Of course, it's perfectly reasonable to assume that someone only got a few second- or third-choice votes, and that they'd only be identified as those in front of them on the ballots are eliminated. In that case, the answer is basically the same, but we need to make sure the map gets populated with all candidate names, not just those with a first place vote somewhere. In some ways, the code is even simpler, but it requires two passes over the list instead of just one.

```
static string identifyLeastPopular(ballot *ballots) {
   Map<string, int> counts;
   for (ballot *curr = ballots; curr != NULL; curr = curr->next) {
      for (int i = 0; i < curr->votes.size(); i++) {
         counts[curr->votes[i]] = 0;
      }
   }

   for (ballot *curr = ballots; curr != NULL; curr = curr->next) {
      counts[curr->votes[0]]++;
   }

   int threshold = INT_MAX; // another approach to establishing a threshold
   string leastPopular;
   foreach (string candidate in counts) {
      if (counts[candidate] < threshold) {
         leastPopular = candidate;
         threshold = counts[candidate];
      }
   }

   return leastPopular;
}
```

This question was given as a CS106B exam question just under a year ago, and we were content with either interpretation.

b.

```
static void eliminateLeastPopular(ballot *& ballots, const string& name) {
   ballot *curr = ballots;
   while (curr != NULL) {
      for (int i = 0; i < curr->votes.size(); i++) {
         if (curr->votes[i] == name) {
            curr->votes.remove(i);
            break; // assume no one ever gets two votes on same ballot
         }
      }

      ballot *next = curr->next;
      if (curr->votes.isEmpty()) {
         // wire up neighboring nodes
         if (next != NULL) next->prev = curr->prev;
         if (ballots == curr) ballots = next;
         else curr->prev->next = next;
         delete curr;
      }
      curr = next;
   }
}
```

Lab Problem 1 Solution: Merging Lists

Because we don't have **prev** pointers, this is much trickier than it first appears. It's a simply stated problem, but the solution is far from simple.

One approach is to use tail recursion—even if tail recursion is frowned upon—so you can continually reframe the problem in terms of a less involved request to merge two lists.

```
static node *mergeLists(node *one, node *two) {
   if (one == NULL) return two;
   if (two == NULL) return one;

   // got this far? then neither list is empty
   if (one->value <= two->value) {
      one->next = mergeLists(one->next, two);
      return one;
   } else {
      two->next = mergeLists(one, two->next);
      return two;
   }
}
```

The above is very clean, easy to understand, and a totally reasonable answer in an academic setting. I've mentioned several times, however, that tail recursion is generally verboten if it's easy to reframe the implementation in terms of iteration. (As with the **braid** example above, do you really want a stack of 10,000 recursive calls to accumulate to merge two lists of length 5,000? What about the 1,000,000 recursive calls needed to merge two lists of length 500,000 each?)

The iterative solution is, of course, tail recursion free, but it requires an advanced understanding of pointers in order to get the job done. I'm presenting this here as another example of a function that uses double pointers as an update-by-reference technique. It's intimidating, but worth the 15 minutes it'll take you to draw the memory diagrams needed to understand it.

```
static node *mergeLists(node *one, node *two) {
   node *merge = NULL;
   node **mergep = &merge;

   while (one != NULL && two != NULL) {
      if (one->value <= two->value) {
         *mergep = one;
         one = one->next;
      } else {
         *mergep = two;
         two = two->next;
      }
      mergep = &((*mergep)->next);
   }

   if (one != NULL) *mergep = one;
   else *mergep = two;
   return merge;
}
```