CS106X Handout 28S
Autumn 2019 October 30th-31st, 2019

Section Solution

Problem 1 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(const Vector<skipListNode *>& heads, int key) {
   const Vector<skipListNode *> *levels = &heads;
   int level = levels->size() - 1;

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

   return false;
}
```

Problem 2 Solution: Ranked Choice Voting

a.

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

   int threshold = -1;
   string leastPopular;
   for (const string& candidate: 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 many ways, the code is even simpler, but it requires two passes over the list instead of just one.

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

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

   int threshold = INT_MAX; // another approach to establishing a threshold
   string leastPopular;
   for (const string& candidate: counts) {
      if (counts[candidate] < threshold) {
         leastPopular = candidate;
         threshold = counts[candidate];
      }
   }

   return leastPopular;
}
```

This question was given as an exam question seven or so years 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;
   }
}
```

Problem 3 Solution: Binary Tree Synthesis

a)

```
static treeNode *listToBinaryTree(const listNode *head) {
   if (head == NULL) return NULL;
   treeNode *root = new treeNode;
   root->value = head->value;
   root->left = listToBinaryTree(head->next);
   root->right = listToBinaryTree(head->next);
   return root;
}
```

b)

```
static treeNode *listToBinaryTree(const listNode *head) {
   treeNode *root;
   Queue<treeNode **> children;
   children.enqueue(&root);

   for (const listNode* curr = head; curr != NULL; curr = curr->next) {
      int numChildren = children.size();     // take a snapshot of the size
      for (int i = 0; i < numChildren; i++) {
         treeNode **nodep = children.dequeue();
         *nodep = new treeNode;
         (*nodep)->value = curr->value;
         children.enqueue(&(*nodep)->left);
         children.enqueue(&(*nodep)->right);
      }
   }

   // everything in Queue points to what needs to be NULLed out
   while (!children.isEmpty()) {
      treeNode **nodep = children.dequeue();
      *nodep = NULL;
   }

   return root;
}
```