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.

```cpp
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.

```cpp
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 some ways, the code is even simpler, but it requires two passes over the list instead of just one.

```cpp
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 six or seven years ago, and we were content with either interpretation.

b.

```cpp
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;
}

Problem 4 Solution: Cartesian Trees

I gave this question as an exam question about nine years ago, and most did very well on it. There are several approaches, but the most straightforward is one which scans the sequence of interest and identifies the minimum element (and its location), establishes that as the root, and then recurs on either side, as with:

    static int findIndexOfMinimum(const Vector<int>& inorder, int low, int high) {
        int index = low;
        for (int i = low + 1; i <= high; i++) {
            if (inorder[i] < inorder[index]) {
                index = i;
            }
        }
        return index;
    }
static node *arrayToCartesianTree(const Vector<int>& inorder, int low, int high) {
        if (low > high) return NULL;
        int index = findIndexOfMinimum(inorder, low, high);
        node *root = new node;
        root->value = inorder[index];
        root->left = arrayToCartesianTree(inorder, low, index - 1);
        root->right = arrayToCartesianTree(inorder, index + 1, high);
        return root;
    }

static node *arrayToCartesianTree(const Vector<int>& inorder) {
    return arrayToCartesianTree(inorder, 0, inorder.size() - 1);
}

Lab Problem 5 Solution: Finding Words In Character Trees
As is always the case, there are many ways to get this to work. I’m just presenting the solution I came up with when I came up with this problem a few years ago.

static bool suffixExists(const node *tree, const string& suffix) {
        if (suffix.empty()) return true;
        if (tree == NULL || suffix[0] != tree->ch) return false;
        return suffixExists(tree->left, suffix.substr(1)) ||
                suffixExists(tree->right, suffix.substr(1));
    }

static bool wordExists(const node *tree, const string& str) {
        if (str.empty()) return true; // the empty tree contains the empty string
        if (tree == NULL) return false;
        if (suffixExists(tree, str)) return true;
        return wordExists(tree->left, str) || wordExists(tree->right, str);
    }