Section Solution

Discussion Problem 1 Solution: Tree Rotations

a)

```
static void rotateLeft(node **parentp) {
   node *parent = *parentp;
   node *rightChild = parent->right;
   parent->right = rightChild->left;
   rightChild->left = parent;
   *parentp = rightChild;
}
```

b)

```
static void pullToRoot(node **rootp, int value) {
   node *root = *rootp;
   if ((root == NULL) ||
       (root->value == value)) return; // no reason to rotate
   if (root->value < value) { // value would be in right subtree
      pullToRoot(&root->right, value);
      rotateLeft(rootp);
   } else {
      pullToRoot(&root->left, value);
      rotateRight(rootp);
   }
}
```

Discussion Problem 2 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;
}
```

Lab Problem 1 Solution: Cartesian Trees

I gave this question as an exam question about two 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(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(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;
}

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

Lab Problem 2 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) return false;
   if (suffix[0] != tree->ch) return false;
   return suffixExists(tree->left, suffix.substr(1)) ||
          suffixExists(tree->right, suffix.substr(1));
}

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);
}
```