

Section Solution

Discussion Problem 1 Solution: Tree Rotations

a)

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

b)

```
void pullToRoot(node **rootp, int value) {
    node *root = *rootp;
    if ((root == NULL) ||
        (root->value == value)) return; // no rotating to be done
    if (root->value < value) { // value would be in right subtree
        pullToRoot(&(root->right), value);
        leftRotate(rootp);
    } else {
        pullToRoot(&(root->left), value);
        rightRotate(rootp);
    }
}
```

Discussion Problem 2 Solution: Binary Tree Synthesis

a)

```
treeNode *listToBinaryTree(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)

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

    for (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:

```

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

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;
}

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;
}

```

Lab Problem 2 Solution: Exponential Trees

The add operation is certainly the most complex of the three methods, which is why it's the most interesting one to implement. I'm providing a version of **add** that uses tail recursion, since that's the approach the reader takes while inserting values into a binary search tree. An equally good approach—and in my opinion even better—would be to implement it iteratively to avoid the recursion altogether. But, the iterative solution makes use of double pointers, which can make the implementation substantially more difficult to implement, debug, and understand.

```

int ExponentialTree::find(Vector<string>& values, string value) {
    for (int i = 0; i < values.size(); i++) {
        if (value < values[i])
            return i;
    }

    return values.size();
}

void ExponentialTree::add(node *& root, string value, int depth) {
    if (root == NULL) {
        root = new node;
        root->depth = depth;
        root->values.add(value);
        root->children = NULL;
        return;
    }

    int insertionPoint = find(root->values, value);
    if (root->values.size() < root->depth) {
        root->values.insertAt(insertionPoint, value);
        return;
    }

    if (root->children == NULL) {
        root->children = new node *[depth + 1];
        for (int i = 0; i <= depth; i++) {
            root->children[i] = NULL;
        }
    }

    add(root->children[insertionPoint], value, depth + 1);
}

void ExponentialTree::add(string value) {
    return add(root, value, 1);
}

```