Solution 1: Linked Lists

a. [5 points] My solution uses a node **, though we’re perfectly fine if you use prev and curr pointers as our initial linked list examples did.

```c
struct node {
    int value;
    node *next;
};

static bool contains(node *& list, int value) {
    node **currp = &list;
    while (*currp != NULL and (*currp)->value != value) {
        currp = &(*currp)->next;
    }
    bool found = *currp != NULL;
    if (found && *currp != list) {
        node *curr = *currp;
        *currp = curr->next;
        curr->next = list;
        list = curr;
    }
    return found;
}
```

**Problem 1a Criteria:**
- Correctly crawls the list to find the matching node: 1 point
- Correctly compiles the information needed to splice a matching node out: 1 point
- Properly rewires the matching node’s predecessor and successor to be neighbors: 1 point
- Correctly rewires the matching node to lead the list: 1 point
- Properly handle the situation where the matching node is at the front: 1 point (it’s possible code that doesn’t need to be run is fine when it runs)

b. [5 points] This function is trickier than it might seem, because you need to not only build the reverse linked list, but you also need to identify the very last next field of the original list and update it to address the reverse.

```c
static void mirror(node *list) {
    node *reverse = NULL;
    for (node *curr = list; curr != reverse; curr = curr->next) {
        node *n = new node;
        n->value = curr->value;
        n->next = reverse;
        reverse = n;
        if (curr->next == NULL)
            curr->next = reverse;
    }
}
```

**Problem 1b Criteria:**
- Correctly visits every node in the original list: 1 point
- Correctly allocates a new node for every value in the original and populates its value field: 1 point
- Properly wires the accumulation of new nodes to be the reverse of the original list: 1 point
- Properly concatenates the reverse to the original just as everything finishes
- Correctly handles both the empty list and the singleton list (ideally without special casing): 1 point
Solution 2: Trie Insertion Trace

a. [5 points] This is the more interesting half of the problem, because it’s clear how the first version breaks down. (The new nodes in both part a and part b have uninitialized bools, but I draw them as false, since the trie node we relied on in class had a constructor and set the isWord bool to false. We were equally happy with false or with questions marks.

```cpp
def ensureNodeExists1(node *root, const string& str, int pos = 0) {
    if (root == NULL) root = new node;
    if (pos == str.size()) return root;
    node *child = root->suffixes[str[pos]];
    return ensureNodeExists1(child, str, pos + 1);
}
```

All str variables at all levels refer to same string.

Problem 2a Criteria:
- Properly display variables like pos and str: 1 point
- Properly initializes all child values to be copies of pointers.
- Properly initializes all roots to be copies of child (and then overwrites it if NULL)
- Properly allocates nodes: 1 point
- Properly inserts missing letters and maps them to NULL.
Solution 2: Trie Insertion Trace [continued]

b. [5 points] As the problem statement implied this version worked as expected, you shouldn’t be surprised very much of the diagram below.

```cpp
node *ensureNodeExists2(node * & root, const string & str, int pos = 0) {
    if (root == NULL) root = new node;
    if (pos == str.size()) return root;
    node * & child = root->suffixes[str[pos]];
    return ensureNodeExists2(child, str, pos + 1);
}
```

All `str` variables at all levels refer to same string.

Problem 2b Criteria:
- Properly display variables like `pos` and `str`: 1 point
- Properly initializes all child values to reference map values: 1 point
- Properly initializes all roots to reference whatever child references: 1 point
- Properly allocates nodes: 1 point
- Properly inserts missing letters and ultimately has each map to new nodes: 1 point
Solution 3: All Things Tree

a. [7 points]

```cpp
static void contract(node *& root) {
    if (root == NULL) return;
    contract(root->left);
    contract(root->right);
    if ((root->left == NULL && root->right == NULL) ||
        (root->left != NULL && root->right != NULL)) return;

    node *child = root->left;
    if (child == NULL) child = root->right;
    delete root;
    root = child;
}
```

**Problem 3a Criteria:**
- Identifies the NULL base case (the original tree may be empty, so it's necessary): 1 point
- Recursively contracts the left and right subtrees: 2 points
- Correct returns without surgery if the root was a leaf or was full: 1 point
- Correctly identifies the child that should be hoisted up a lever: 1 point
- Correctly rewires the tree around the one-child node: 1 point
- Correctly levies the `delete` call against the removed node: 1 point

b. [8 points]

```cpp
static Set<node *> construct(int low, int high) {
    Set<node *> trees;
    if (high < low) {
        Set<node *> trees;
        trees += NULL;
        return trees;
    }

    for (int divider = low; divider <= high; divider++) {
        Set<node *> lefts = construct(low, divider - 1);
        Set<node *> rights = construct(divider + 1, high);
        for (node *left: lefts) {
            for (node *right: rights) {
                node *root = new node;
                root->value = divider;
                root->left = cloneTree(left);
                root->right = cloneTree(right);
                trees.add(root);
            }
        }
    }
    return trees;
}
```

```cpp
static Set<node *> construct(int n) {
    return construct(1, n);
}
```

**Problem 3b Criteria:**
- Properly reframes the primary call to be a wrapper function with lower and upper bounds: 1 point
- Properly handles the empty-range situation by returns a singleton set that’s the empty tree: 1 point
- Correctly considers every single value in the range [low, high] as root values, including low and high: 1 point
- For each choice of divider, recursively constructs the set of all legal binary search trees that can hang to the left and right: 2 point
- Properly allocates a new node for each left-right pairing and embeds copies of divider, left, and right: 2 points
- Properly adds each tree to the set and ultimately returns it: 1 point