CS106X Midterm Examination Solution

Solution 1: Correct and Incorrect Tree Insertion [8 points]

Assume the following binary tree node definition:

```
struct node {
   int value;
   node *left, *right;
};
```

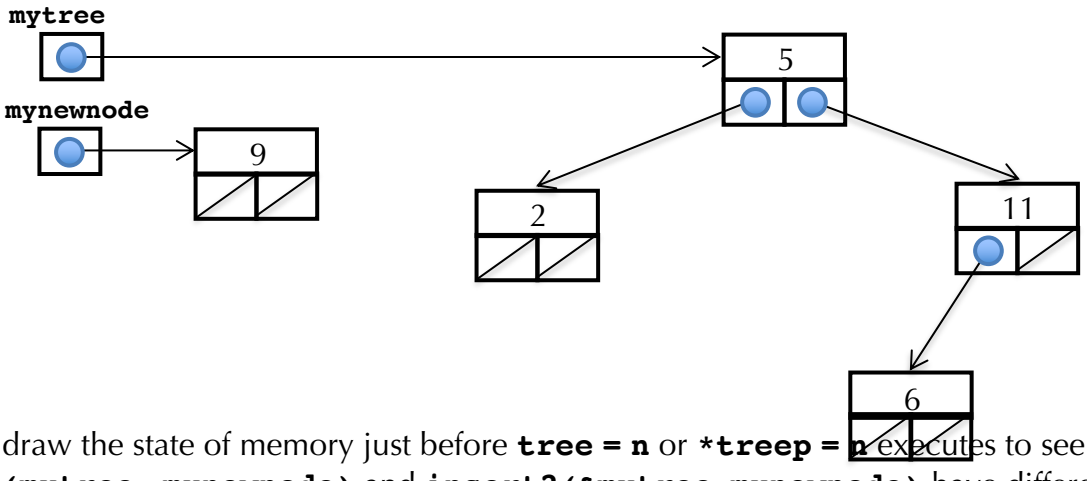
and consider the following two recursive implementations:

```
static void insert1(node *tree, node *n) {
   if (tree == NULL) {
      tree = n;
      return;
   }

   if (tree->value < n->value)
      insert1(tree->right, n);
   else
      insert1(tree->left, n);
}
```

```
static void insert2(node **treep, node *n) {
   if (*treep == NULL) {
      *treep = n;
      return;
   }

   node *tree = *treep;
   if (tree->value < n->value)
      insert2(&tree->right, n);
   else
      insert2(&tree->left, n);
}
```

Presumably, each exists with the intent to hanging the node addressed by **`n`** along the fringe of the binary search tree rooted at **`tree`** and **`*treep`**, respectively, so that the binary search tree property is maintained. One of them works, and the other does not.

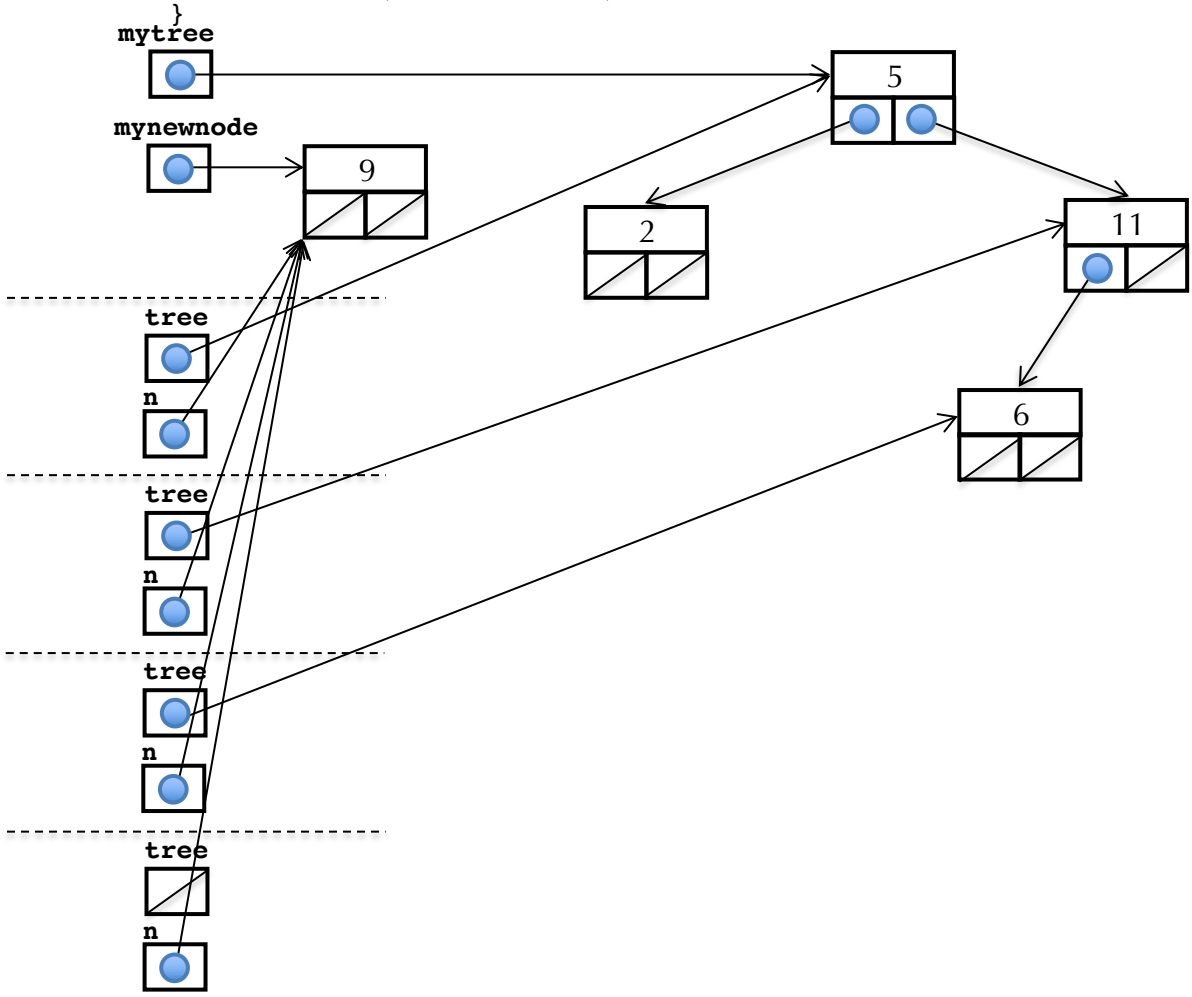
Assume the following illustration captures exactly how **`mytree`** and **`mynewnode`** (each of type **`node *`**) have been initialized just prior to a call to one of the two functions above:



You're to draw the state of memory just before **`tree = n`** or **`*treep = n`** executes to see why **`insert1(mytree, mynewnode)`** and **`insert2(&mytree, mynewnode)`** have different behaviors.

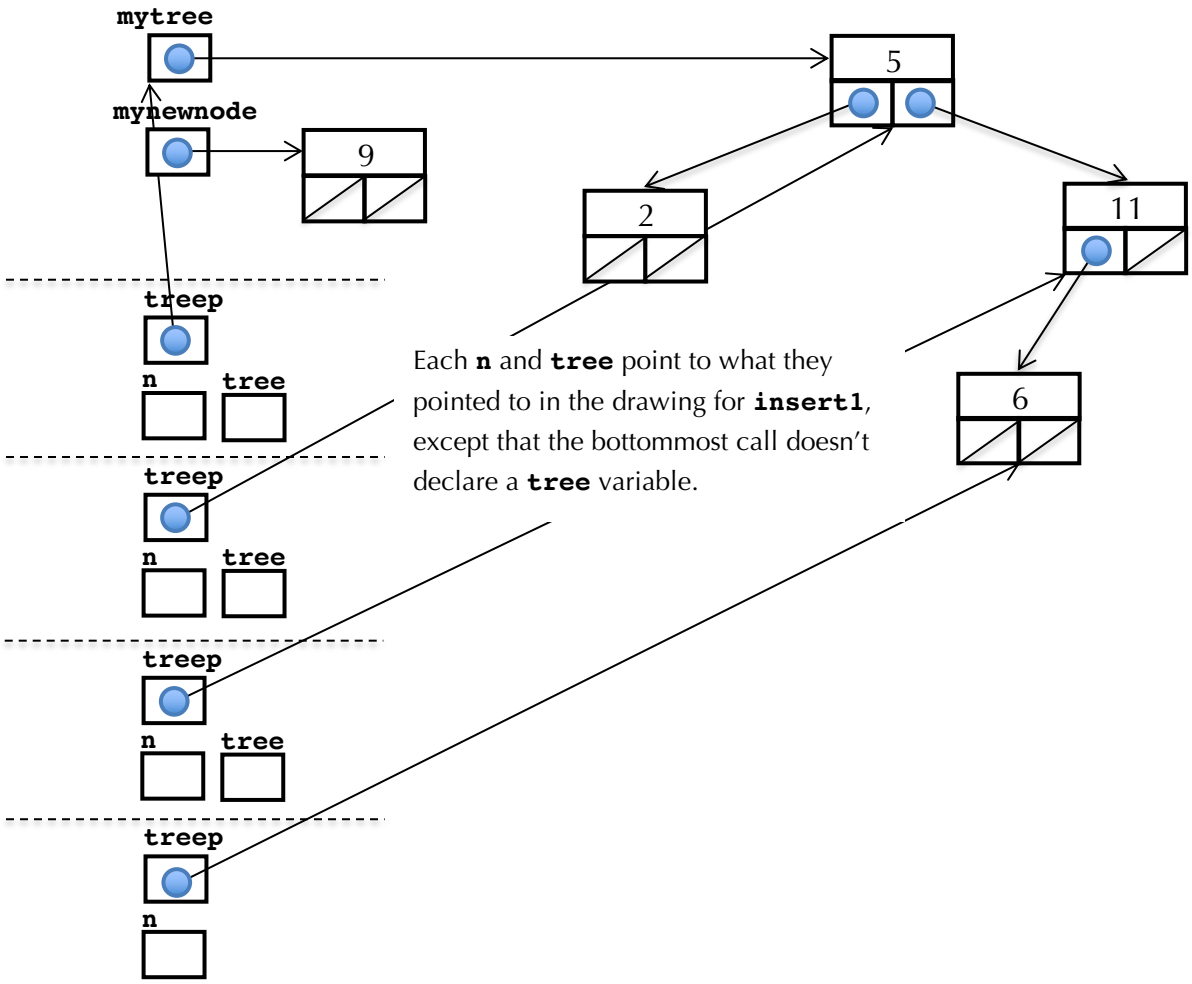
a. [4 points] Draw the state of memory as the call to **`insert1(mytree, mynewnode)`** bottoms out, just before its **`tree = n`** statement executes. You'll want to draw all of the parameters for all function calls, being clear what each of the parameters associated with each of the recursive calls contains.

```
static void insert1(node *tree, node *n) {
   if (tree == NULL) {
      tree = n;
      return;
   }

   if (tree->value < n->value)
      insert1(tree->right, n);
   else
      insert1(tree->left, n);
}
```



b. [4 points] Do the same thing, this time for **`insert2(&mytree, mynewnode)`**. Draw the state of memory just before the **`*treep = n`** statement executes. As before, you'll want to draw all of the parameters (and the local variable) for all recursive calls.

```
static void insert2(node **treep, node *n) {
   if (*treep == NULL) {
      *treep = n;
      return;
   }

   node *tree = *treep;
   if (tree->value < n->value)
      insert2(&tree->right, n);
   else
      insert2(&tree->left, n);
}
```



Solution 2: Permutations and Linked Cycles [10 points]

Consider the following permutation:

6	3	8	5	4	1	0	7	2
0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**

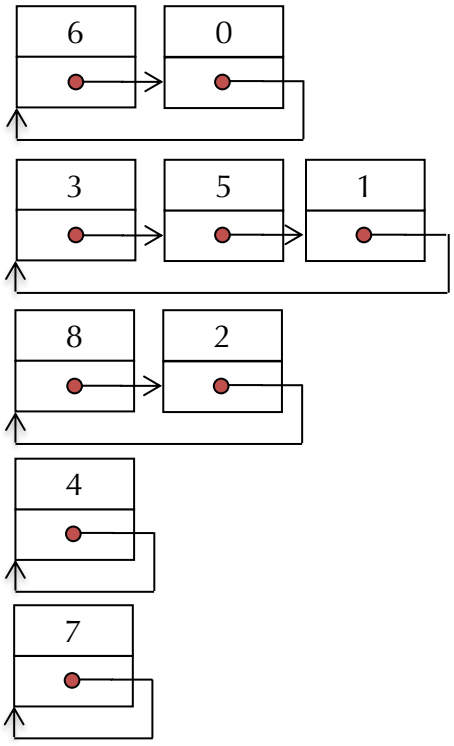
The above sequence is clearly a permutation on the numbers 0 through 8, inclusive. But it is also a **partition** of the first 9 nonnegative integers into 5 cycles, and those five cycles are:

$$6 \rightarrow 0,\ 3 \rightarrow 5 \rightarrow 1,\ 8 \rightarrow 2,\ 4, \text{ and } 7$$

The $3 \rightarrow 5 \rightarrow 1$ cycle is implied by the permutation: The 5 is at index 3 and therefore follows the 3, the 1 is at index 5 and therefore follows the 5, and the 3 is at index 1 and therefore follows the 1. Specifically, every number **`k`** in the permutation is part of exactly one cycle, and **`k`**'s successor in the cycle is at index **`k`** of the permutation.

Note that 8 and 2 are mutual successors of one another, since the 2 is at index 8 and the 8 is at index 2. That means 2 follows 8 follows 2 in another cycle separate from $3 \rightarrow 5 \rightarrow 1$. And 4 is in its own cycle, since it resides at index 4 and is its own successor. Cycles of size 1 are completely legitimate.

Each cycle can be represented as a circular linked list, as in the following diagrams:



Assume you have access to the following type definition:

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

Implement the **permutationToCycles** function to process the supplied permutation and return a **Vector<node *>** of all of its cycles. The address stored by each **node *** in the return value can be that of any one of the nodes in a given cycle, and each cycle is represented as a circular, singly linked list. You may assume that the referenced **Vector<int>** is nonempty and holds some permutation of the numbers 0 through **v.size()** - 1, inclusive.

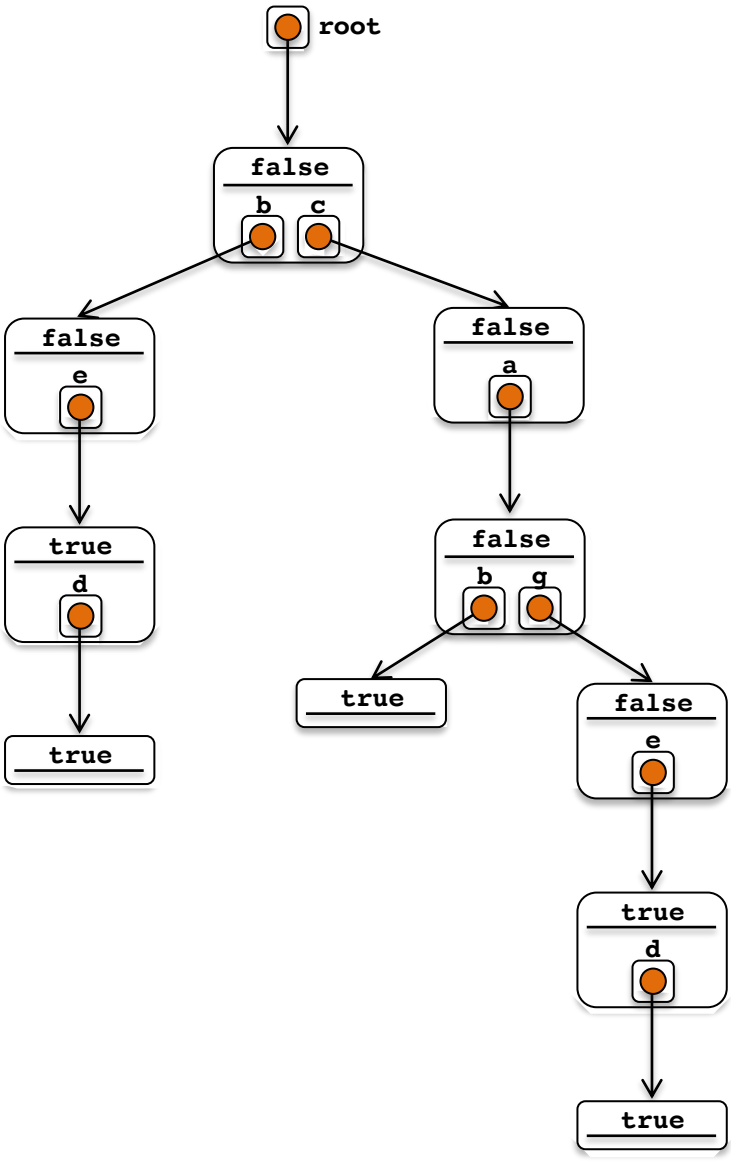
Use the rest of this page to complete the implementation.

```
static Vector<node *> permutationToCycles(const Vector<int>& permutation) {
   Vector<node *> nodes;
   for (int i = 0; i < permutation.size(); i++) {
      node *n = new node;
      n->value = permutation[i]; // NULL'ing next not necessary in my implementation
      nodes += n;
   }

   for (int i = 0; i < nodes.size(); i++) {
      nodes[i]->next = nodes[permutation[i]];
   }

   Vector<node *> cycles;
   Set<int> catalogued;
   for (int i = 0; i < nodes.size(); i++) {
      if (!catalogued.contains(nodes[i]->value)) {
         node *curr = nodes[i];
         cycles += curr;
         while (!catalogued.contains(curr->value)) {
            catalogued += curr->value;
            curr = curr->next;
         }
      }
   }

   return cycles;
}
```

Solution 3: Tries and Shared Suffixes [12 points]

The **trie**, discussed in Handout 28, effectively relies on this **node** definition:

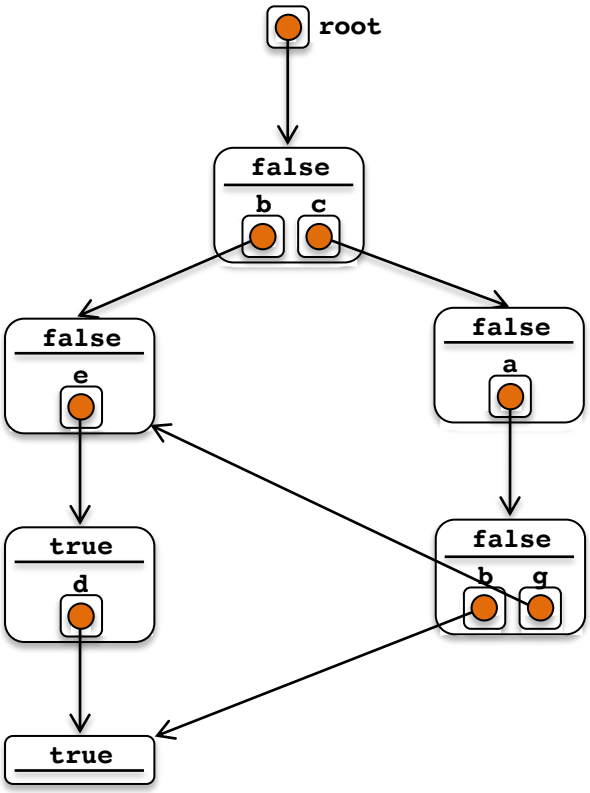
```
struct node {
   bool isWord;
   Map<char, node *> suffixes;
   node() {
      isWord = false;
   }
  ~node() {
      for (char ch: suffixes)
         delete suffixes[ch];
   }
};
```



The words **be**, **bed**, **cab**, **cage**, and **caged** are all stored in the trie on the right. Note, however, that many of the sub-tries are duplicates of one another. The three leaf nodes, for instance, are all structurally identical, as they all contain **true**s and have empty **suffixes Map**s. And the **b** of **be** and **bed** addresses a sub-trie that is structurally identical to that addressed by the **g** of **cage** and **caged**. If we can identify two or more sub-tries as structurally identical, we can reduce the memory footprint of a trie by merging them—that is, rather than allowing multiples copies of identical sub-tries to exist, we can retain exactly one as the master copy, rewire all pointers to the duplicate sub-tries to instead address the master, and dispose of the duplicates.

This problem will lead you through a series of illustrations and functions that will ultimately allow you to rid of duplicate sub-tries without impacting the set of encoded words and prefixes.

a. [1 point] Draw the trie from the previous page in the space below, but don't include any duplicate sub-tries, and rewire pointers that address the roots of duplicates to instead address the root of the retained master.



b. [6 points] Implement the **operator<** predicate function that returns **true** if the trie rooted at **one** is "less than" the trie rooted at **two**. Be sure that whenever **operator<(one, two)** returns **true** it's the case that **operator<(two, one)** returns **false**, and that **operator<(one, two)** and **operator<(two, one)** each return **false** if and only if the two tries are structurally identical. You're otherwise free to impose whatever rules you want to decide whether one trie is "less than" another. This function should be implemented recursively.

```
static bool operator<(const node *one, const node *two) { // assume both non-NULL
   if (one->isWord != two->isWord)
      return !one->isWord;

   if (one->suffixes.size() != two->suffixes.size())
      return one->suffixes.size() < two->suffixes.size();

   string oneKeys, twoKeys;
   for (char ch: one->suffixes) oneKeys += ch;
   for (char ch: two->suffixes) twoKeys += ch;

   if (oneKeys != twoKeys)
      return oneKeys < twoKeys;

   for (char ch: one->suffixes) {
      if (one->suffixes[ch] < two->suffixes[ch]) return true;
      if (two->suffixes[ch] < one->suffixes[ch]) return false;
   }

   return false;
}
```

c. [5 points] Finally, implement the **compress** function, which accepts the address of a trie's root, removes and disposes of all duplicate sub-tries, and rewires the pointers to removed duplicates to instead address the roots of retained masters. The fact that you implemented **operator<** from part b means that **node *** can be the keys of a **Map**. In particular, you should implement **compress** recursively (using a wrapper function), and in doing so you'll create and maintain a **Map<node** *, **node *>** (where each key maps to itself) to keep track of all the masters.

```
static void compress(node *& root, Map<node *, node *>& masters) {
   if (masters.containsKey(root)) {
      node *found = masters[root];
      delete root;
      root = found;
      return;
   }

   masters[root] = root;
   for (char ch: root->suffixes)
      compress(root->suffixes[ch], masters);
}

static void compress(node *& root) {
   if (root == NULL) return;
   Map<node *, node *> masters;
   compress(root, masters);
}
```