CS106B
Summer 2013

Handout #20
August 5, 2013

Section Handout 6

Based on handouts by Jerry Cain

Problem One: Listing Words in a Trie

Suppose that you have the following struct representing a node in a trie:

```
struct Node {
      bool isWord;
      Node* children[26];
};
```

In class, we saw how to determine whether a given word exists within a trie. What would we do if we wanted to list off all the words in a trie?

Write a function

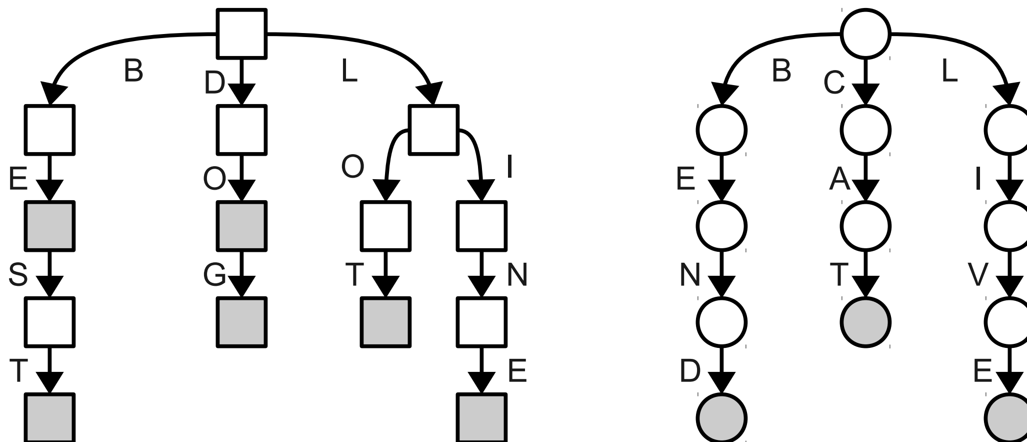
```
void allWordsIn(Node* root, Vector<string>& result);
```

that accepts as input a pointer to the root node of a trie, then populates the **`Vector`** argument with a list of all strings contained within the trie.

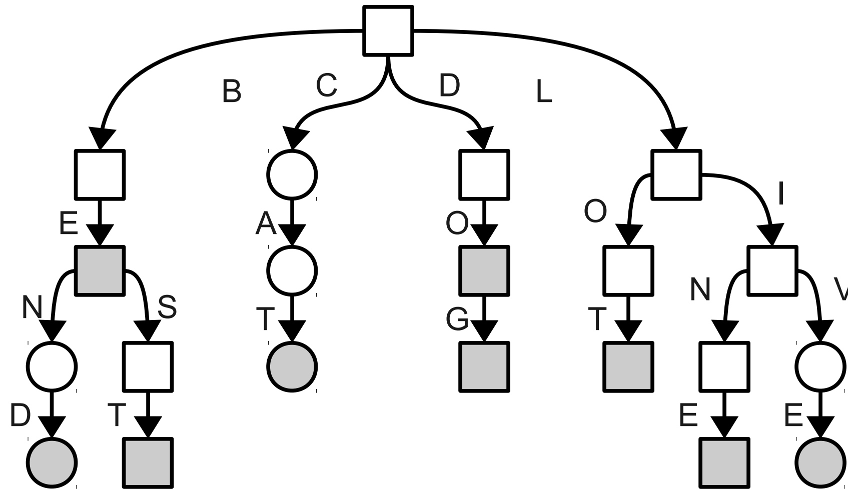
Problem Two: Trie Merging

Suppose that you have two tries, each storing some set of words. You are interested in constructing one new trie containing all of the words stored collectively within those two tries. One way to do this would be to use the above **`allWordsIn`** function to compute the set of all words in both of the tries, then to insert those words one at a time into a new trie. While this approach works correctly, it is not particularly efficient.

A better approach would be to take the two existing tries and rewire the nodes within those tries to combine them together into one single trie. For example, given these two tries:



We could splice the nodes together as follows to produce a new trie:



Write a function

```
Node* mergeTries(Node* first, Node* second);
```

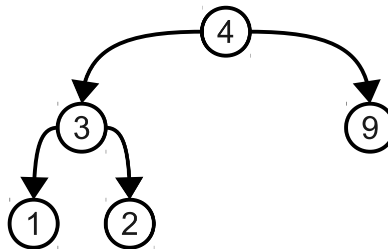
that accepts as input pointers to two different tries, merges those tries together, then returns a pointer to the root of the new trie. In doing so, your should be sure to free the memory for all nodes that were not required in the merged trie.

Problem Three: Checking BST Validity

Suppose that you have the following structure representing a node in a binary search tree of integers:

```
struct Node {
      int value;
      Node* left;
      Node* right;
};
```

You are given a pointer to a **Node** that is the root of some type of binary tree. However, you are not sure whether or not it is a binary *search* tree. That is, you might have a tree like this one:



which is not a valid binary search tree.

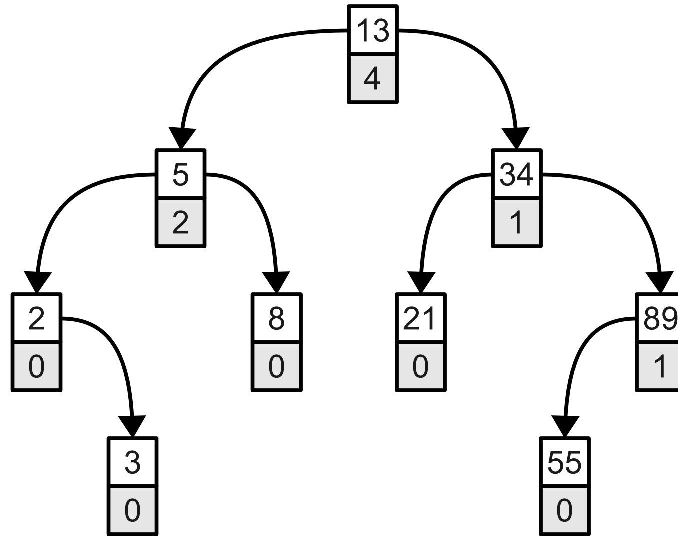
Write a function

```
bool isBST(Node* root);
```

that, given a pointer to the root of a tree, determines whether or not that tree is a legal binary search tree.

Problem Four: Order Statistic Trees

An *order statistic tree* is a binary search tree where each node is augmented with the number of nodes in its left subtree. For example, here is a simple order statistic tree:



Suppose that you have the following struct representing a node in an order statistic tree:

```
struct Node {
      int value;
      int leftSubtreeSize;
      Node* left;
      Node* right;
};
```

Write a function

```
Node* nthNode(Node* root, int n);
```

that accepts as input a pointer to the root of the order statistic tree, along with a number n, then returns a pointer to the node in the tree that holds the nth smallest value in the tree (zero-indexed). If n is negative or at least as large as the number of nodes in the tree, your function should return **NULL** as a sentinel.