Section Solutions 6

Problem One: Listing Words in a Trie

One possible implementation is to recurse over the trie, tracking the prefix of the words we've built up so far. When we reach a node corresponding to a word, we can output that prefix as a word in the trie.

```
void allWordsIn(Node* root, Vector<string>& result) {
    return recAllWordsIn(root, result, "");
}

void recAllWordsIn(Node* root, Vector<string>& result, string soFar) {
    /* Base case: If the root is NULL, there are no words here. */
    if (root == NULL) return;

    /* If the current node is a word, then the prefix so far must be a valid word
     * in the trie.
     */
    if (root->isWord) result += soFar;

    /* Recursively explore all children. */
    for (int i = 0; i < 26; i++) {
        recAllWordsIn(root->children[i], result, soFar + char('A' + i));
    }
}
```

Problem Two: Trie Merging

This function has a beautiful recursive structure to it. If both tries are empty, we're done. If there is exactly one trie, we can trivially merge it with the other. Otherwise, we combine the two tries together recursively, pick one of them as our resulting trie, and then deallocate the other.

```
Node* mergeTries(Node* first, Node* second) {
    /* Base case: If both tries are empty, their union is the empty trie. */
    if (first == NULL && second == NULL) return NULL;

    /* Base case: If exactly one trie is NULL, the result is the other trie. */
    if (first == NULL) return second;
    if (second == NULL) return first;

    /* Recursive step: Combine these nodes together.  We'll arbitrarily decide to
     * store the result in first.
     */

    /* If either node marks a word, the result is a word. */
    first->isWord = first->isWord || second->isWord;

    /* Recursively merge all 26 children. */
    for (int i = 0; i < 26; i++) {
        first->children[i] = mergeTries(first->children[i], second->children[i]);
    }

    /* Free memory from the second trie. */
    delete second;
    return first;
```

```
}
```

Problem Three: Checking BST Validity

There are many ways to solve this problem. One option is to do an inorder walk of the tree, keeping track of the last node visited. The value of the current node should always be at least the value that came before it:

```
bool isBST(Node* root) {
    Node* lastNode = NULL;
    return recIsBST(root, lastNode);
}

/* Checks if the current node represents a valid BST, given that the last node
 * visited was prev.
 */
bool recIsBST(Node* root, Node*& prev) {
    /* Base case: If the tree is empty, it's a valid BST. */
    if (root == NULL) return true;

    /* Recursive step: Check if the left subtree is valid.  If so, then check if
     * our value comes next in sequence, then check the right subtree.
     */
    if (!recIsBST(root->left, prev)) return false;

    /* Check our value against the previous one, if one exists. */
    if (prev != NULL && root->value <= prev->value) return false;

    /* Mark that we visited this node, then search the right subtree. */
    prev = root;
    return recIsBST(root->right, prev);
}
```

Problem Four: Order Statistic Trees

This function has a beautiful recursive structure to it. Essentially, to look up a value, we compare the current index to how many nodes are in the left subtree. If it's equal, then we have the value we want. If it's less, we descend into the left. Otherwise, we descend into the right.

```
Node* nthNode(Node* root, int n) {
    /* Base case: If the root is NULL, nothing exists. */
    if (root == NULL) return NULL;

    /* Recursive step: If this is the value we're looking for, we're done. */
    if (n == root->leftSubtreeSize) return root;

    /* Otherwise, if this node is to the left, look there. */
    if (n < root->leftSubtreeSize) return nthNode(root->left, n);

    /* Otherwise, look to the right.  Take into account the number of nodes that
     * we skipped when doing so.
     */
    return nthNode(root->right, n - 1 - root->leftSubtreeSize);
}
```