**Problem 1: Encoding General Trees**

A general tree is one where each node has an arbitrary number of children. Here’s an example:

![General Tree Diagram]

It’s possible to encode an arbitrary tree in binary tree form by subscribing to a left-child, right-sibling representation. Each node in the binary tree representation has two children. The left child is the first child of the corresponding node in the general tree, and the right child is the right sibling of the corresponding node in the general tree. So, the above would map to the following binary tree structure:

![Binary Tree Diagram]

Note, for example, how all of the children of the root in the original tree now form the right spine on the sub-tree that hangs from the root in the new tree.
Write a function called `encode`, which accepts the root of a general tree and constructs and returns the corresponding binary tree.

```c
struct genTreeNode {
    int value;
    Vector<genTreeNode *> children; // genTreeNode *s are never NULL
};

struct binTreeNode {
    int value;
    binTreeNode *left;  // addresses first child within general tree equivalent
    binTreeNode *right; // addresses right sibling within general tree equivalent
};

binTreeNode *encode(genTreeNode *root);
```

**Problem 2: Patricia Trees**

Consider the following illustration:

What’s drawn above is an example of a Patricia tree—similar to a trie in that each node represents some prefix in a set of words. The child pointers, however, are more elaborate, in that they not only identify the sub-tree of interest, but they carry the substring of characters that should contribute to the running prefix along the way. Sibling pointers aren’t allowed to carry substrings that have common prefixes, because the tree could be restructured so that the common prefix is merged into its own connection. By imposing that constraint, that means there’s at most one path that needs to be explored when searching for any given word.

The children are lexicographically sorted, so that all strings can be easily reconstructed in alphabetical order. When a node contains a `true`, it means that the prefix it represents is also a word in the set of words being represented. [The root of the tree always represents the empty string.]
So, the tree above stores the following words:

    cranium, crazy, go, golf, golfing, goober, peg, perky, petulance, pork, and pundit.

These two type definitions can be used to manage such a tree.

    struct connection {
        string letters;
        struct node *subtree; // will never be NULL
    };
    struct node {
        bool isWord;
        Vector<connection> children; // empty if no children
    };

Implement the containsWord function, which accepts the root of a Patricia tree and a word, and returns true if and only if the supplied word is present. Even though the connections descending from each node are sorted alphabetically, you should just do a linear search across them to see which one, if any, is relevant.

    static bool containsWord(const node *root, const string& word);

Problem 3: Regular Expressions

Regular expressions are, for the purposes of this problem, comprised of lowercase alphabetic letters along with the characters *, +, and ?. In these regular expressions, the lowercase letters match themselves. * is always preceded by an alphabetic character and matches zero or more instances of the preceding letter. + is similar to *, except that it matches 1 or more instances of the preceding letter. ? states that the preceding letter may or may not appear exactly once. Here are some examples of these regular expressions:

    grape              matches grape as a word and nothing else
    letters?           matches letter and letters, but nothing else
    a?b?c?             matches a, b, c, ab, ac, bc, abc, and the empty string
    lolz*              matches lol, lolz, lolzz, lolzzz, and so forth
    lolz+              matches lolz, lolzz, lolzzz, and so forth

All of the *, + and ? characters must be preceded by lowercase alphabetic letters, or else the regular expression is illegal.

Regular expressions play nicely with the trie data structure we discussed in lecture last week. We’ll use this exposed data structure to represent the trie:

    struct node {
        bool isWord;
        Map<char, node *> suffixes;
    };

Write the matchAllWords function, which takes a trie of words (via its root node address) and a regular expression as described above, and populates the supplied
**Set<string>,** assumed to be empty, with all those words in the trie that match the regular expression.

```cpp
static void matchAllWords(const node *trie, const string& regex, Set<string>& matches);
```

**Problem 4: Dictionaries and Ternary Search Trees**

The **Dictionary** class is a specialized data structure storing all of the English words along with their definitions. Because many words have multiple definitions, each word maps not to a single **string** but a **Vector** of them.

The **Dictionary** is backed by a data structure called a **ternary search tree**. Ternary search trees are hybrids of two data structures we’ve studied extensively over the past few lectures: binary search trees, and tries. Binary trees are space efficient in that the amount of memory used is proportional to the number of entries it stores. Tries are exceptionally fast, because the time to look up, insert, or delete any single word is bounded by the length of its longest word. Ternary search trees combine elements of the two. Like binary search trees, they are space efficient, except that its nodes have three children instead of two. Like tries, they proceed character by character during a search.

A search compares the current character in the key to the letter embedded in a node. If the current character is less, the search continues along the **less** pointer. If the search character is greater, the search follows the **greater** pointer. If the characters match, then the search carries on via the **equal** pointer, but proceeds to the next character in the key.

Here’s the header file for the TST-backed **Dictionary**:

```cpp
class Dictionary {
public:
    Dictionary() { root = NULL; } // inline the obvious implementation
    ~Dictionary();
    void add(const std::string& word, const std::string& definition);

private:
    struct node {
        char letter;
        Vector<std::string> *definitions;
        node *less, *equal, *greater;
    };
    node *root;
};
```

If the string represented by a particular node is a word in the **Dictionary**, then that node's **definitions** field stores the address of a dynamically allocated **Vector<string>** to store the definitions in the order they were inserted. If the string represented by a particular node is not itself a word but rather a prefix of one or more words, then that node’s **definitions** field stores **NULL**.

You’re to implement the one **public** method and the destructor.
Before you get started on the code, let’s be clear about what the TST-backed Dictionary would look like if the words "pig", "cow", "cop" and "cozy" were inserted, in that order.

Note that the node surrounding the last letter of a word is the one that stores the address of the dynamically allocated Vector<string>.

a. Present your implementation of the add method, which ensures that the specified word gets added if it isn’t already, and appends the specified definition (even if it’s a duplicate) to the end of its Vector of definitions. Make sure you properly allocate and initialize any nodes that need to be incorporated, and be sure to properly allocate space for the Vector<string> whenever a word is inserted for the very first time.
void Dictionary::add(const string& word, const string& definition);

b. Now implement the destructor to properly dispose of all dynamically allocated memory that’s been allocated over the course of the Dictionary’s lifetime.

Dictionary::~Dictionary();