Section Handout

Problem 1: 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.

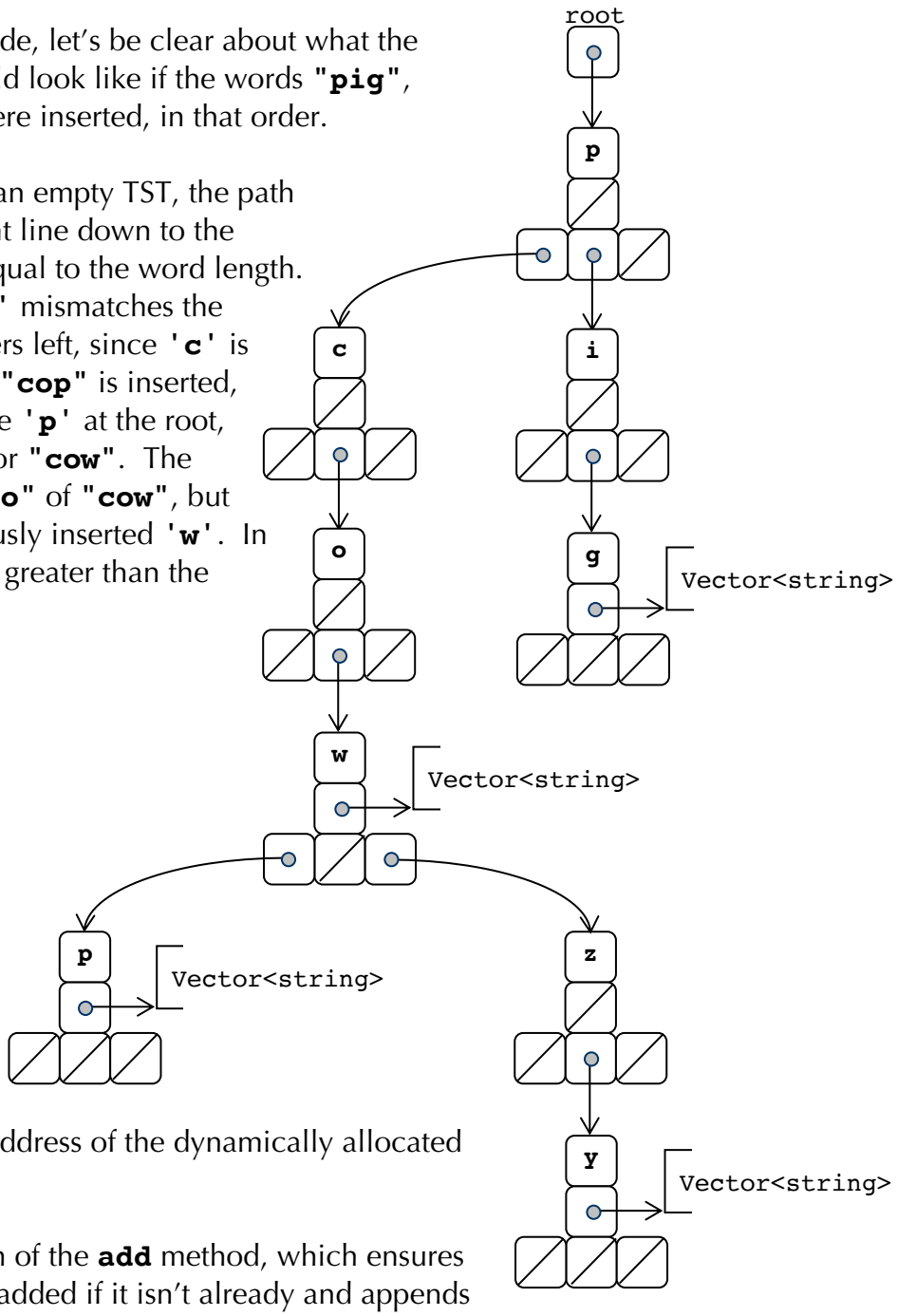
Here's the header file for the TST-backed **`Dictionary`**:

```
class Dictionary {
public:
   Dictionary() { root = NULL; } // inline the obvious implementation
   ~Dictionary();

   void add(const string& word, const string& definition);

private:
   struct node {
      char letter;
      Vector<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.



Since **`"pig"`** was inserted into an empty TST, the path inserted on its behalf is a straight line down to the fringe, and that path length is equal to the word length. When **`"cow"`** is inserted, its **`'c'`** mismatches the **`'p'`** at the root, so insertion veers left, since **`'c'`** is less than **`'p'`**. Similarly, when **`"cop"`** is inserted, The leading **`'c'`** mismatches the **`'p'`** at the root, So insertion veers left as it did for **`"cow"`**. The **`"co"`** of **`"cop"`** matches the **`"co"`** of **`"cow"`**, but the **`'p'`** mismatches the previously inserted **`'w'`**. In general, the path length is often greater than the length of the string it represents.

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();
```

Problem 2: 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 the preceding letter may appear 0 or 1 times. 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`**, **`lolzz`**, 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.

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

Problem 3: People You May Know

Because Facebook is interested in growing out its social graph, users are often presented with a list of other users they might be friends with even though that friendship isn't officially recorded. That list is drawn from the set of Facebook users who are strictly two degrees away from you—that is, the list of your friends' friends that aren't already friends with you.

Assume that the following node definition is used to represent a Facebook user:

```
struct user {
   int userID;          // unique
   string name;         // not necessarily unique
   Set<user *> friends; // assume friendship is symmetric
};
```

a. Write a function called **getFriendsOfFriends**, which given the address of your node in the social graph, returns as a **Set** the collection of nodes representing those on Facebook who are two degrees away from you. (Assume the logged in user is 0 hops away from him or herself and shouldn't be included.)

```
static Set<user *> getFriendsOfFriends(user *loggedinuser);
```

b. [Credit: Zach Birnholz] Now write the more general **getKthDegreeFriends** function that given the address of a user node and an integer k returns the set of Facebook users who are precisely k hops away.

```
static Set<user *> getKthDegreeFriends(user *loggedinuser, int k);
```

Problem 4: Detecting Cycles

Given access to a graph (in the form of the exposed **graph** introduced in lecture), write a predicate called **containsCycle**, which returns **true** if there are any cycles whatsoever in the graph, and **false** otherwise. The ability to detect cycles, and the ability to confirm that the addition of an edge doesn't introduce cycles, is important for some applications (e.g. your final assignment, Stanford 1-2-3, needs to confirm that no two cell formulas mutually depend—directly or eventually—on each other, and C++ compilers sometimes elect to check that no two header files mutually **#include** one an other).

```
static bool containsCycle(graph& g);
```