

CS106X Practice Midterm

Exam Facts:

When: Wednesday, November 30th from 7:00 – 10:00 p.m.
Where: Gates B01, with overflow to Gates B03.

Note this practice exam is longer than your midterm will be.

Coverage

The midterm covers everything up through trees (binary search trees, Cartesian trees, tries, Patricia trees, etc). I'll be starting up on graphs today, but you won't see anything graph-like on your exam. The exam will emphasize material not tested on the first midterm, so expect to be drilled on pointers, dynamically allocated memory, linked lists, hashing and hash tables, mapping (in the **mapA11** sense), and all things trees. Understand going in that issues pertaining to & versus * versus . versus -> are important details that matter.

Some changes between this midterm and your first one:

- This midterm will be three hours long instead of two.
- Graded midterms will be distributed to the section leaders, and your section leader will return your exam during dead week's discussion section.

Problem 1: Linked Structures

a. Write a function called **concatenateMaps**, which accepts a linked list of **Map<string>**s, and returns a single map whose key set is the union of all of the keys of all the maps in the list. The value bound to each key in the return map is determined as follows:

- If the key only appears in one of the maps in the list, then its value should become the same key's value in the constructed map being returned.
- If the key appears two or more times, then the key's values in the constructed map should be the ordered concatenation of all of the corresponding values in the list.

Your implementation should not change any of the maps in the list.

```
struct node {
    Map<string> map;
    node *next;
};

Map<string> concatenateMaps(node *maplist);
```

b. Write a function **stretchList** that takes a nonempty linked list of integers and stretches it so that the first element is replicated one additional time, the second is replicated two additional times, the third is replicated three additional times, and so forth. As an example, the following list:

$$3 \rightarrow 4 \rightarrow 1 \rightarrow 5 \rightarrow 1$$

would be transformed into

$$3 \rightarrow 3 \rightarrow 4 \rightarrow 4 \rightarrow 4 \rightarrow 1 \rightarrow 1 \rightarrow 1 \rightarrow 1 \rightarrow 5 \rightarrow 5 \rightarrow 5 \rightarrow 5 \rightarrow 5 \rightarrow 1 \rightarrow 1 \rightarrow 1 \rightarrow 1 \rightarrow 1 \rightarrow 1$$

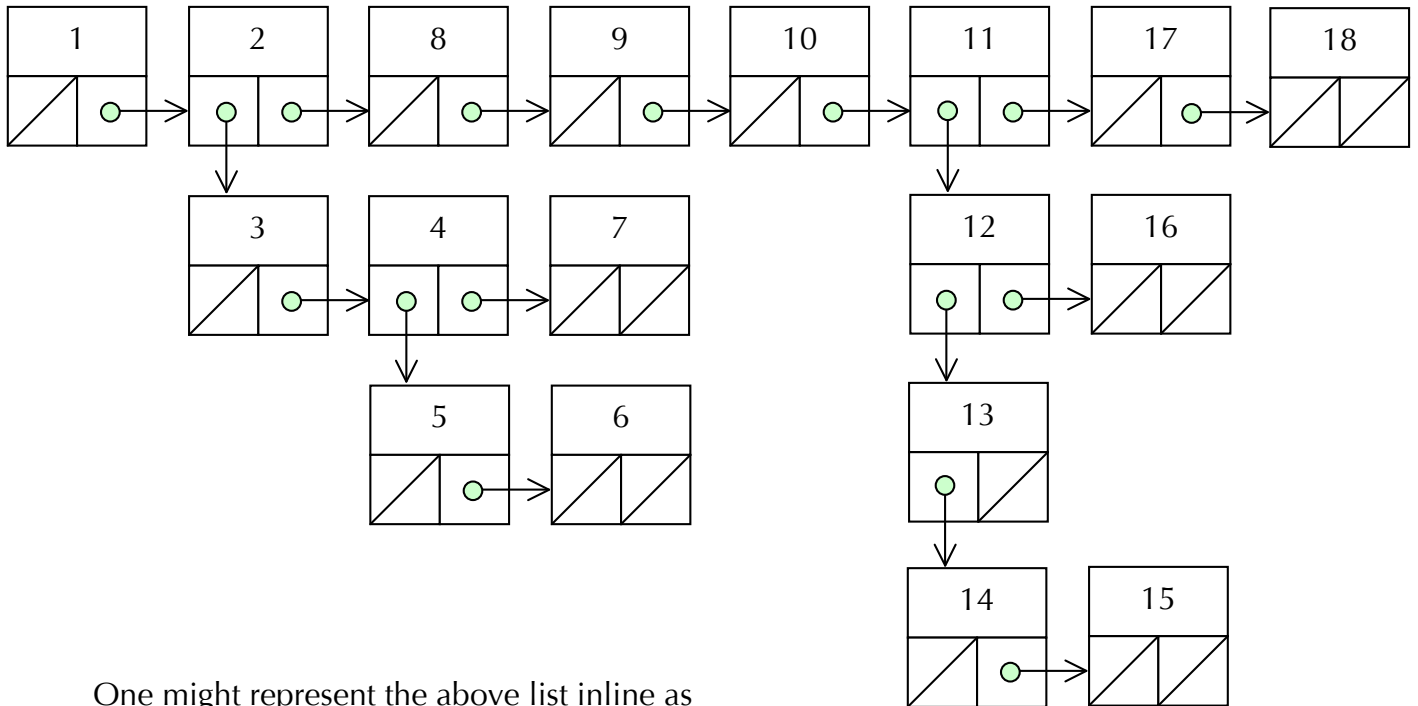
```
struct node {
    int value;
    node *next;
};

void stretchList(node *list);
```

c. Consider the following linked list node definition:

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

The node definition is the traditional definition, except that each node can store a child list in its **down** field. One such list might look like this:



One might represent the above list inline as

[1 2 [3 4 [5 6] 7] 8 9 10 11 [12 [13 [14 15]] 16] 17 18].

Note: the numbers don't need to be sorted, much less sequential.

Write a function called **flattenList**, which serializes the incoming list to a traditional singly linked list of integers, splicing the flattening of the **down** list in between a number and its successor. As an example, the above list would be transformed into the list

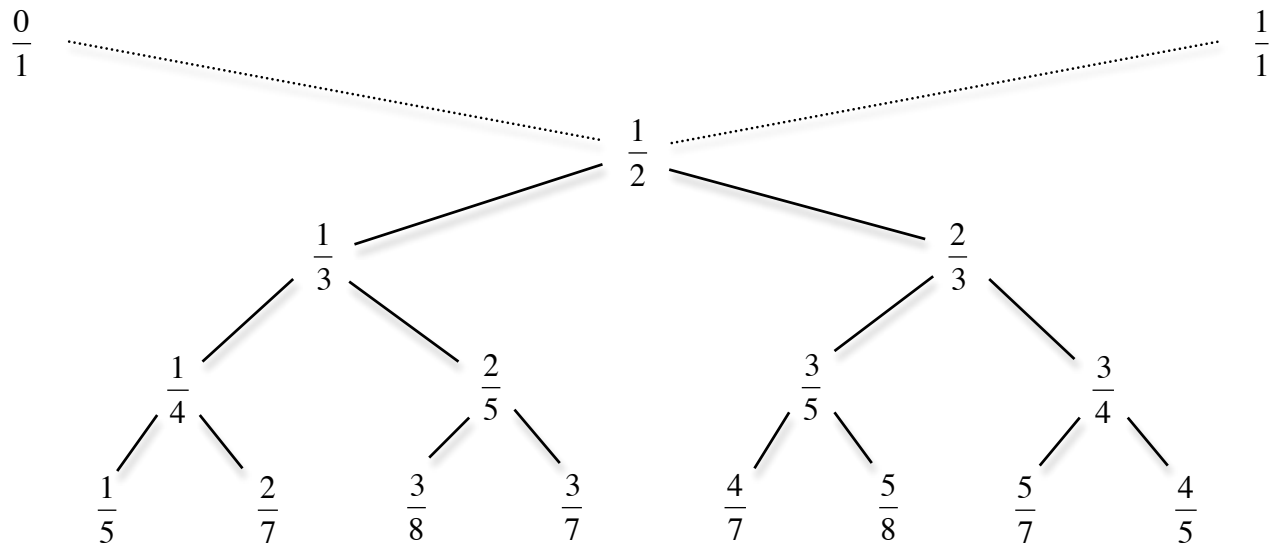
[1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18]

Your implementation shouldn't change the incoming list, but instead should construct a new list (where all down fields are set to **NULL**) to be the linearization of the incoming one.

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

node *flattenList(node *list);
```

- d. Recall the Stern-Brocot construction from an earlier section handout, which looked like this:



Assume you've access to the following data structures:

```
struct fraction {
    int numerator;
    int denominator;
};
```

```
struct node {
    fraction value;
    node *left;
    node *right;
};
```

Implement the **generateSternBrocotTree** function, which builds an in-memory version of the tree to include all (and only those) fractions between 0 and 1 with denominators less than or equal to the one provided, and then returns the address of the root. Recall that each fraction is of the form $\frac{n+n'}{d+d'}$, where $\frac{n}{d}$ is the closest ancestor up and to the left, and $\frac{n'}{d'}$ is the closest ancestor up and to the right. Note that the root of the tree houses $1/2$, and the fractions representing 0 and 1 are not included.

```
node *generateSternBrocotTree(int denominator);
```

- e. Recall that the primary data structure we used to build a trie is defined as:

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

Write a function called **pruneLexicon**, which removes all of the words that can't possibly be formed given the provided histogram of lowercase letters. This function might be repeatedly called, for instance, as a game of Scrabble progresses and the list of formable words—initially a very large subset of the English language—dwindles down

significantly enough that it makes sense to shrink the data structure. When removing words, care should be taken to properly dispose of all nodes that are no longer needed, including those representing irrelevant prefixes.

Assume the provided **node *** is the root of a well-formed trie, and that the referenced **Vector<int>** (of length 26) contains the numbers of a's, b's, c's, and so forth available to form words.

Use this and the next page for your implementation:

```
void pruneLexicon(node *& root, Vector<int>& histogram);
```

Problem 2: Planetarium Memory Trace

Analyze the following program, starting with the call to **littledipper**, and draw the state of memory at the one point indicated. Be sure to differentiate between stack and heap memory, note values that have not been initialized, and identify where memory has been orphaned. Feel free to tear this page out, prepare your answers on a separate sheet, and draw out your final diagram on the next page.

```
struct planet {
    double pluto;
    planet *saturn[4];
    planet **earth;
};

void littledipper() {
    planet venus;
    venus.pluto = 93.1;
    venus.saturn[0] = new planet;
    venus.saturn[0]->saturn[3] = &venus;
    venus.saturn[1] = venus.saturn[0];

    planet *mars = &venus;
    mars->saturn[2] = NULL;
    venus.earth = &venus.saturn[1];
    venus.earth[1] = new planet[2];
    for (int i = 0; i < 4; i++) {
        venus.earth[1]->saturn[i] = venus.saturn[i/2 + 1];
    }

    venus.earth[2] = NULL;

    ← Draw the state of memory just prior to the call to littledipper.
}
```

Problem 3: Superheroes Then and Now

Analyze the following program, starting with the call to **elektra**, and draw the state of memory at the two points indicated. Be sure to differentiate between stack and heap memory, note values that have not been initialized, and identify where memory has been orphaned. Feel free to tear this page out, prepare your answers on a separate sheet, and draw out your final diagrams on the next page. (Note: we expect two independent drawings! In particular, you should not destroy the first drawing in order to draw the second.)

```
struct superhero {
    int wonderwoman;
    superhero *isis;
    int *superman[2];
};

void elektra() {
    superhero marineboy[2];
    superhero *ironman;
    ironman = &marineboy[1];
    marineboy[0].wonderwoman = 152;
    marineboy[0].superman[0] = new int[2];
    marineboy[0].superman[1] = &(ironman->wonderwoman);
    ironman->superman[0] = marineboy[0].superman[1];
    ironman->superman[1] = &(marineboy[0].superman[0][1]);
    *(ironman->superman[1]) = 9189;
    marineboy[1].isis = ironman->isis = ironman;
}
```

← First, draw the state of memory just prior to the call to **barbarella**.

```
    barbarella(marineboy[1], ironman->isis);
}

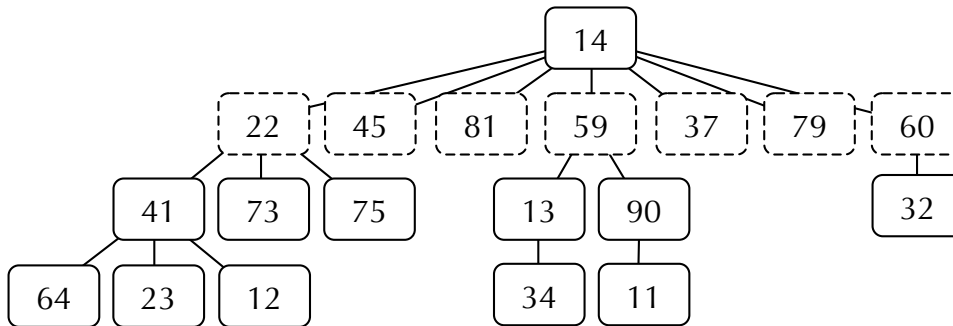
void barbarella(superhero& storm, superhero *& catwoman) {
    storm.wonderwoman = 465;
    catwoman->isis = &storm;
    catwoman->wonderwoman = 830;
    catwoman->isis[0].superman[1] = &(storm.isis->wonderwoman);
    catwoman = &storm;
    catwoman->wonderwoman = 507;
    catwoman->isis = new superhero[2];
}
```

← Second, draw the state of memory just before **barbarella** returns

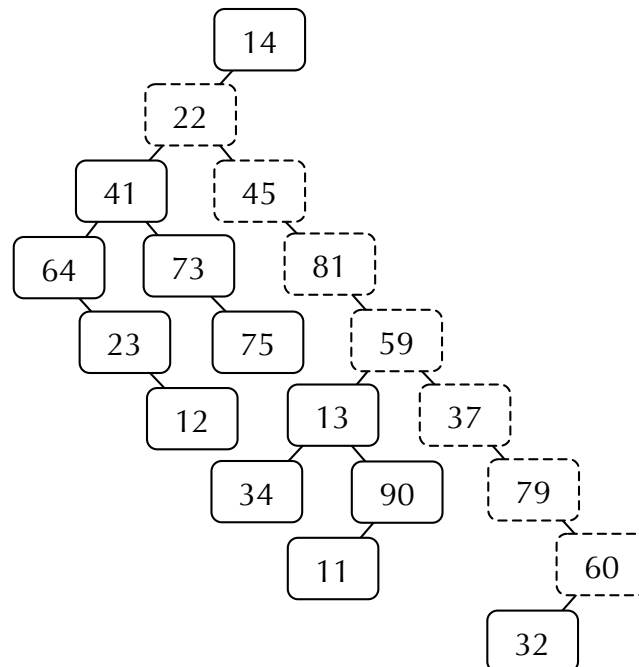
```
}
```

Problem 4: Encoding General Trees

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



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:



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.

```

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

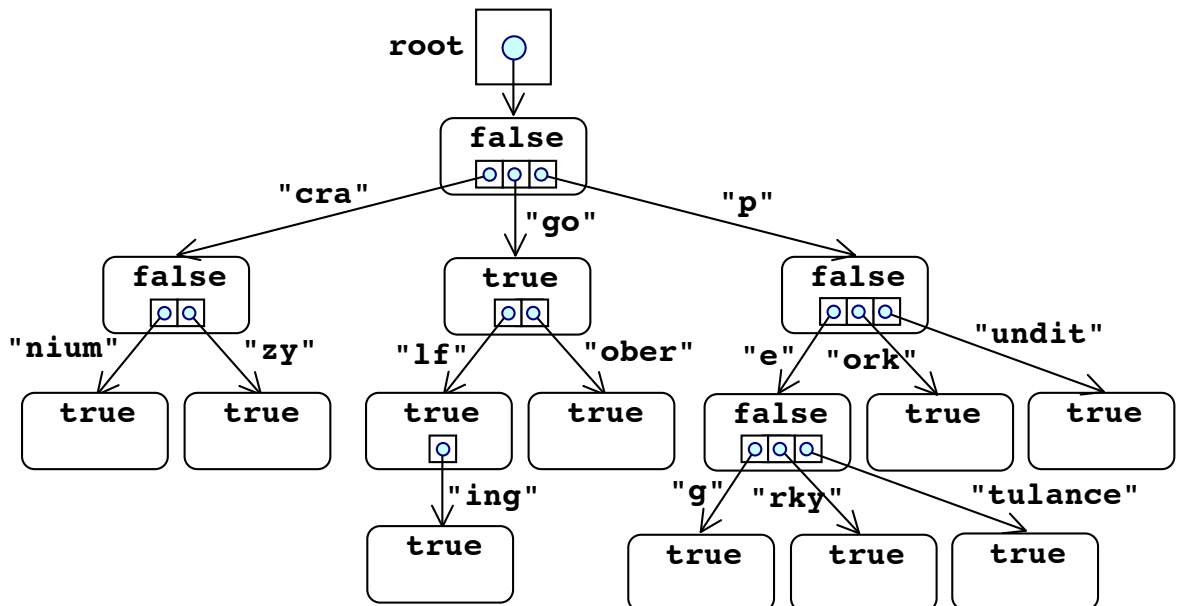
struct binTreeNode {
    int value;
    binTreeNode *left; // addresses first child within general tree equivalent
    binTreeNode *right; // addresses right sibling within general tree equivalent
};

binTreeNode *encode(genTreeNode *root);

```

Problem 5: Patricia Trees Revisited

Let's reconsider the Patricia tree example from this week's section handout, Handout 32.



Recall that a Patricia tree is 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 can 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 actually 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.

The following 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
};
```

The section handout had you implement **containsWord** and **map**, but here you're going to implement the **insertWord** function, which updates the supplied Patricia tree to house the provided word if it doesn't already, all while maintaining the invariant that no two sibling connections ever share a common prefix.

Implement the **insertWord** function, which accepts the root of a Patricia tree and a word and ensures that the word is present and properly represented so that **containsWord** and **map** can find it. (Note: this problem has a lot of cases, and requires a reasonable amount of code be written. Be sure to enumerate all of the different cases before writing any code, and write helper functions to help manage the details.)

```
void insertWord(node *& root, string word);
```

Problem 6: Dictionaries and Ternary Search Trees

The **Dictionary** class is a specialized data structure storing all of the English words along with their definitions. Because some (if not most) 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 two weeks: 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 one string is bounded by the length of the longest word in the dictionary, regardless of the dictionary's size. 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(string word, string definition);
    void mapAll(void (map)(string, Vector<string>& definitions));

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

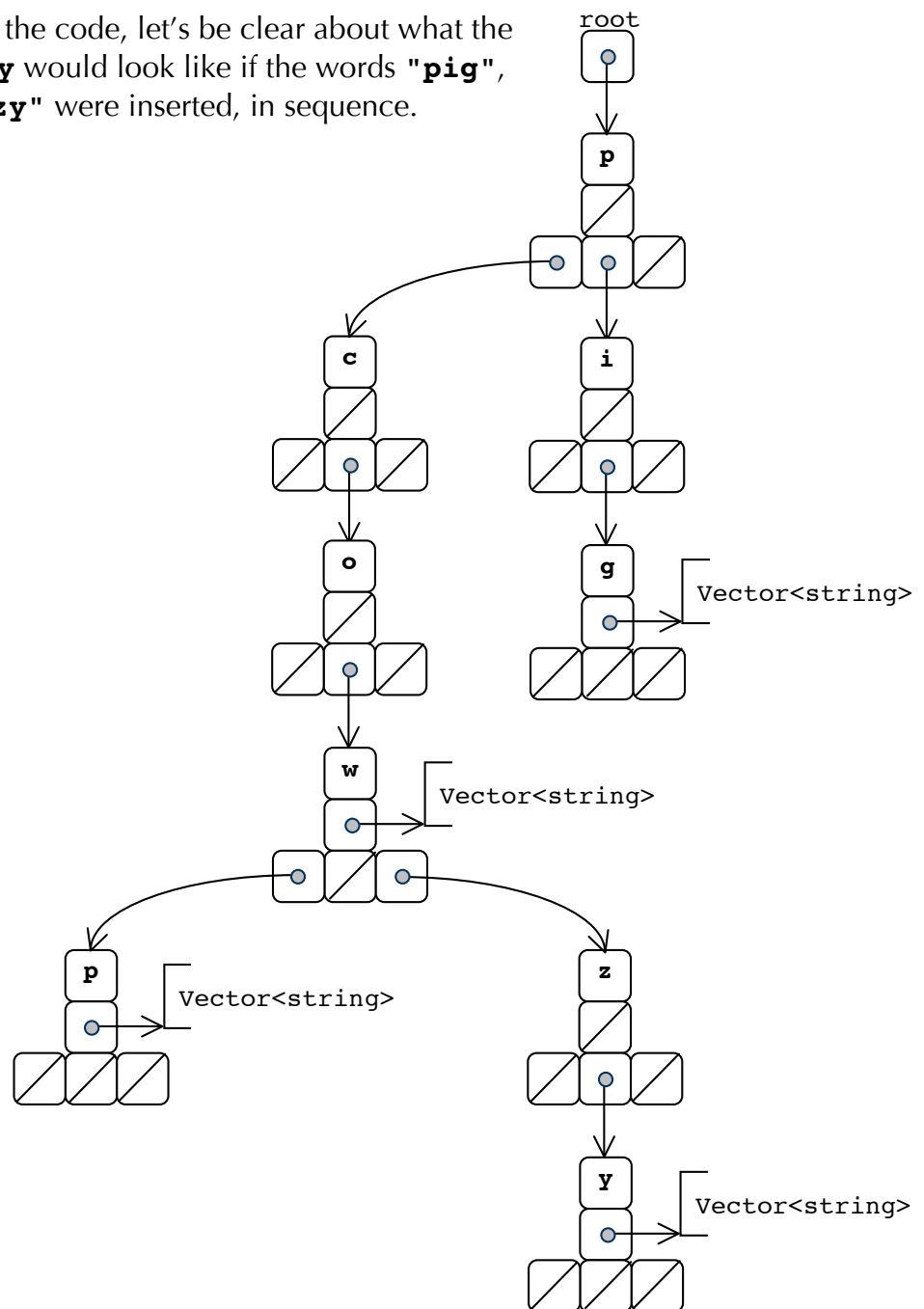
    node *root;
};
```

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**.

Over the course of the next several pages, you're to implement the two **public** methods and the destructor. You're free to write helper methods, but make sure the prototypes of

these helper methods are **crystal clear**. You needn't update the class declaration above—we'll just assume the **private** section would be extended to include your helper method prototypes.

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 sequence.



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

- 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(string word, string definition);
```

- b.) Now present your implementation of the **mapAll** method, which applies the specified function to every single **string-Vector<string>** pair held by the **Dictionary**. Of course, you need to be able to synthesize the strings from the structure of the tree. You also need to ensure that the supplied function is applied to all words in increasing lexicographical order (informally—low to high alphabetically)

```
void Dictionary::mapAll(void (map)(string word, Vector<string>& definitions));
```

- c.) Finally, implement the destructor to properly disposed of all dynamically allocated memory that's been allocated over the course of the **Dictionary**'s lifetime.

```
Dictionary::~Dictionary();
```