CS106X Practice Exam

Exam Facts:

When: Wednesday, November 28th from 7:00 – 10:00 p.m.
Where: Cemex Auditorium, in the new GSB campus.

Coverage

The closed book, closed note, closed computer exam covers everything up through trees (binary search trees, Cartesian trees, tries, Patricia trees, etc). I started 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 exam, so expect to be drilled on pointers, dynamically allocated memory, linked lists, hashing and hash tables, and all things trees. Understand going in that issues pertaining to & versus * versus . versus -> are details that matter very, very much.

Problem 1: Linked Structures

a. Write a function called **`concatenateMaps`**, which accepts a linked list of **`Map<string, 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 in 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, string> map;
   node *next;
};

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

b. Write a function **`stretch`** 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

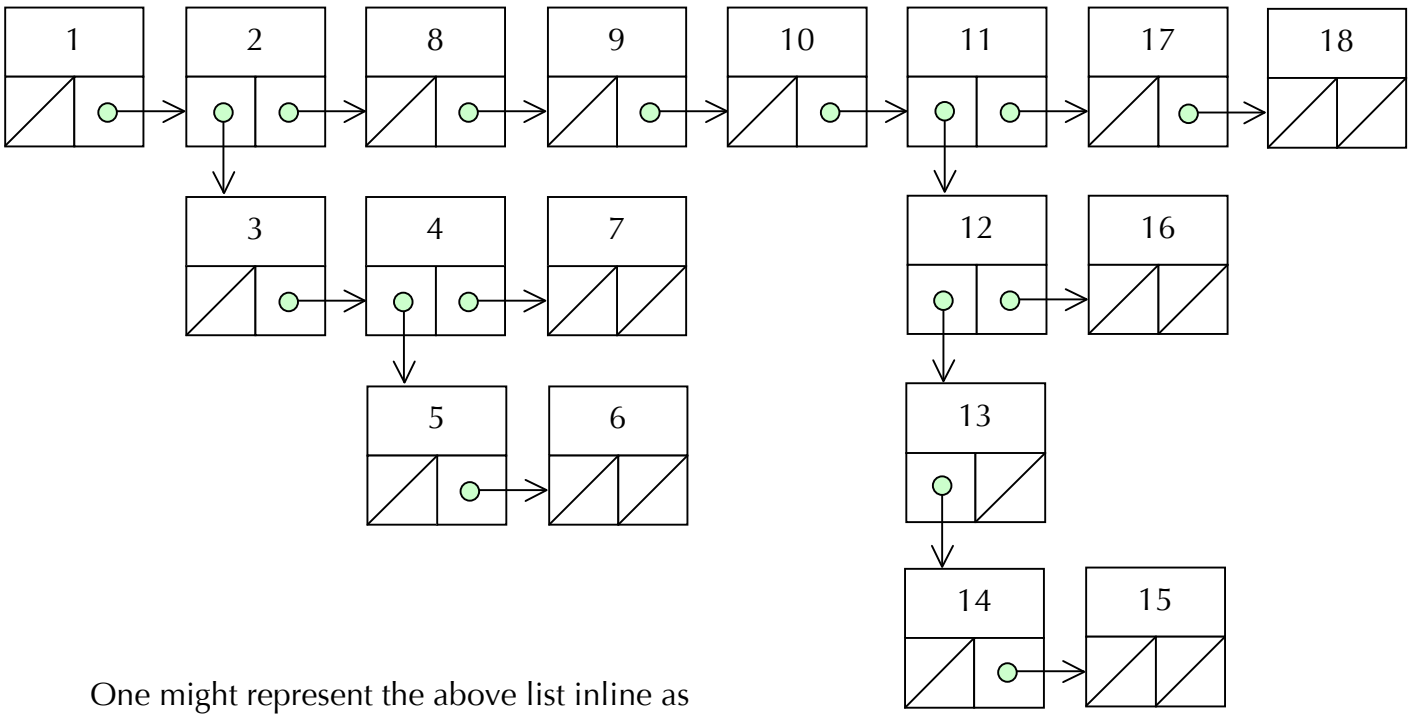
3 → 3 → 4 → 4 → 4 → 1 → 1 → 1 → 1 → 5 → 5 → 5 → 5 → 5 → 1 → 1 → 1 → 1 → 1 → 1

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

static void stretch(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.

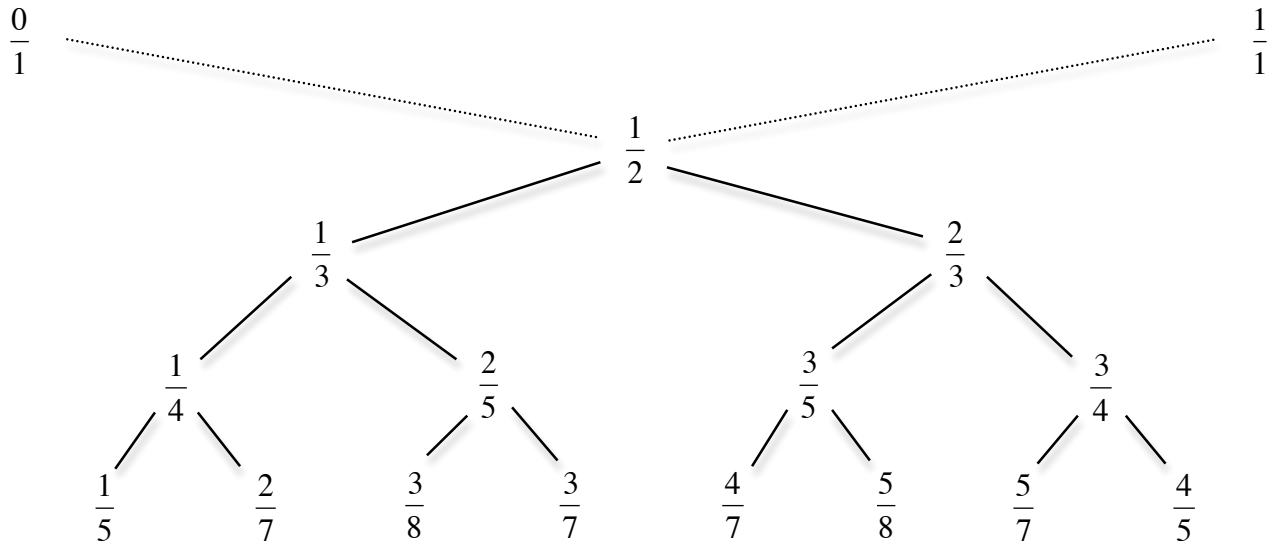
Write a function called **`flatten`**, which serializes the incoming list to a traditional singly linked list of integers, splicing the flattening of the **`down`** list in between a number and its successor. As an example, the above list would be transformed into the list

[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;
};

static node *flatten(const node *list);
```

d. Recall the Stern-Brocot construction from and 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.

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

Problem 2: Phineas and Ferb

Analyze the following code snippet, starting with a call to **`mobilemammal`**, and draw the state of memory at the point indicated. Be sure to differentiate between stack and heap memory, note values that have not been initialized, and identify if and where memory has been orphaned. Draw out your final diagram in the lower half of this page.

```
struct squirrel {
   int spa;
   squirrel *duckymomo[3];
};

void mobilemammal() {
   squirrel phineas[2];
   phineas[1].spa = 100;
   phineas->spa = 777;
   squirrel **ferb = &(phineas[0].duckymomo[1]);
   ferb[0] = ferb[1] = new squirrel;
   ferb[1] = NULL;
   squirrel *pants = *ferb;
   ferb = &(ferb[1]);
   phineas[0].duckymomo[0] = &(phineas[1]);
   phineas[1].duckymomo[0] = phineas[0].duckymomo[0];
   phineas[1].duckymomo[1] = ferb[0];
   phineas[1].duckymomo[2] = pants;
   *pants = phineas[1];
```

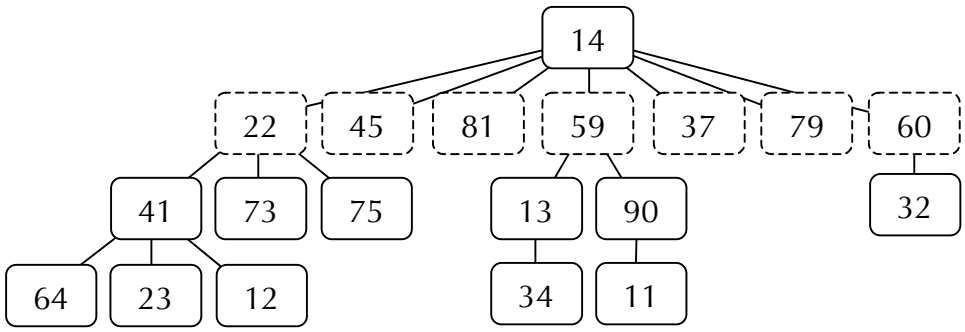
⇐ Draw the state of memory just before `mobilemammal` returns.

```
}
```

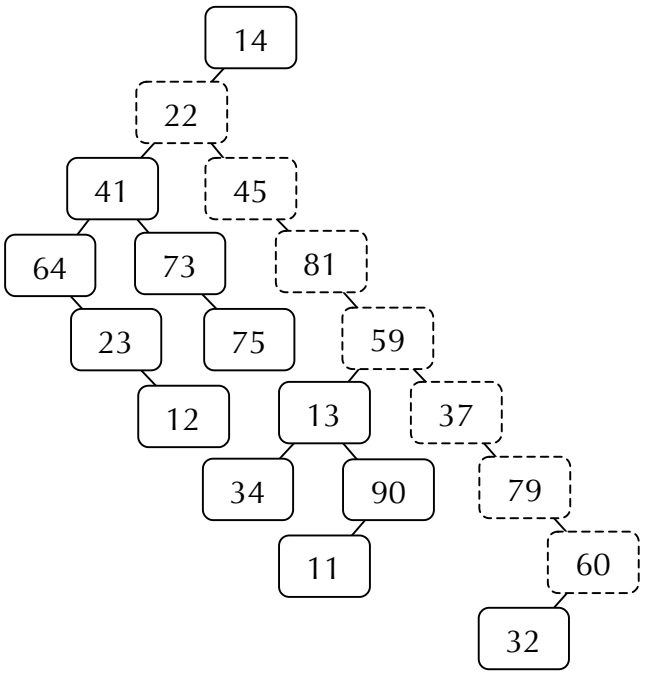
Draw the state of memory just before **`mobilemammal`** returns.

Problem 3: 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 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 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 single word is bounded by the length of the longest word in the dictionary. 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.

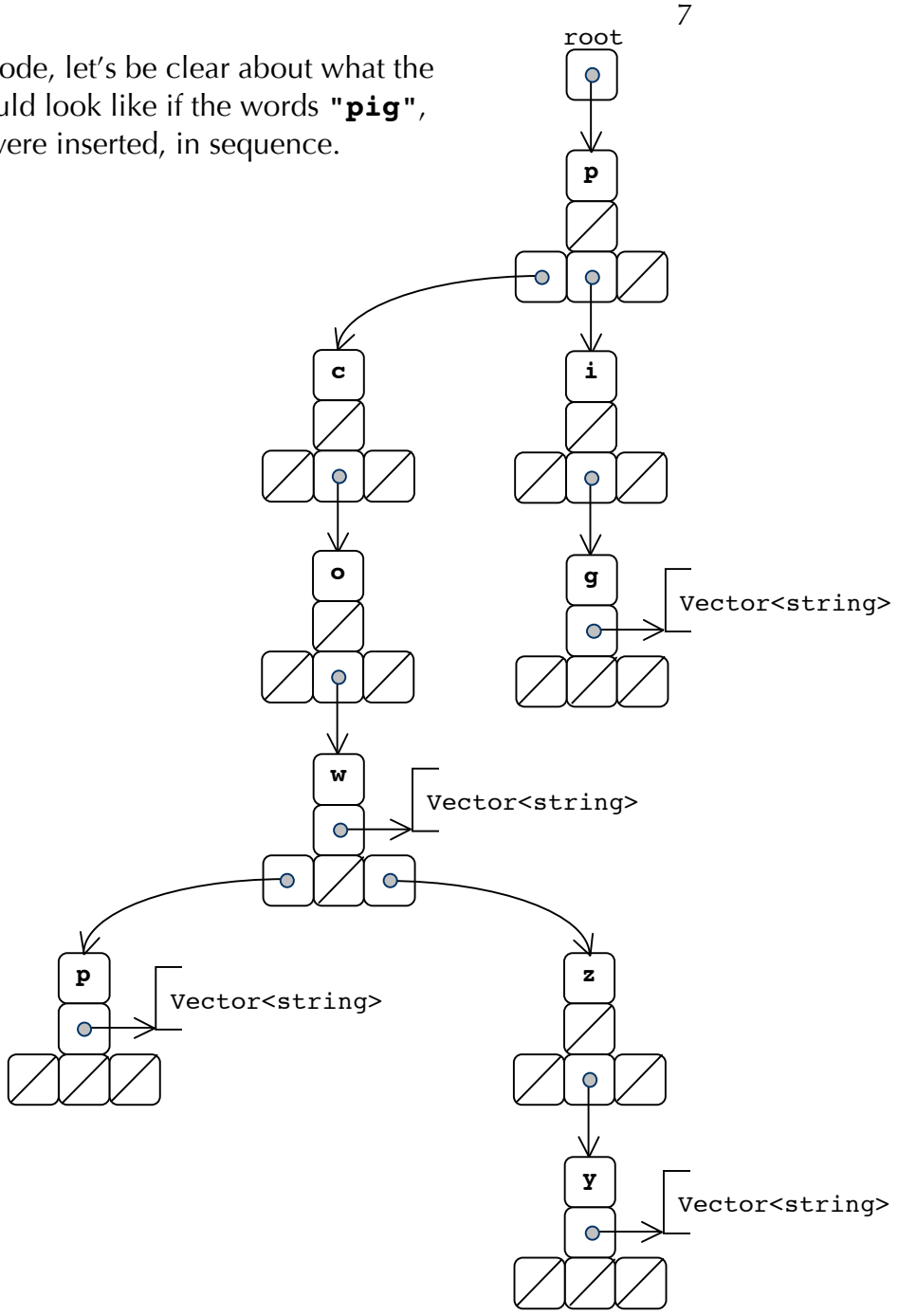
Here's the header file for the TST-backed **Dictionary**:

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

   void add(const 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. 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>`**.

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. Finally, 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();
```