CS106X Midterm Examination

This is an open-book, open-note, closed-electronic-device exam. You have 90 minutes to complete it.

Good luck!

Section Leader:	______________________
Last Name:	______________________
First Name:	______________________
SUNet ID:	______________________**@stanford.edu**

I accept the letter and spirit of the honor code.

(signed) ____

		Score	Grader
1. Linked Lists	[10]	______	______
2. Trie Insertion Traces	[10]	______	______
3. All Things Tree	[15]	______	______
Total	**[35]**	______	______

Problem 1: Linked Lists [10 points]

a. [5 points] Implement a predicate function called **`contains`** that walks a singly linked list for **`value`** and returns **`true`** if and only if the element is found, and **`false`** otherwise. Additionally, if **`value`** is found, then **`contains`** should splice the node storing **`value`** out of the list and prepend it to the front (unless the node was already at the front, in which case it should be left alone). The result is a linked list storing the same information, except the key of interest now resides at the front. If searching for a key is likely to be followed by repeated search for it—in practice, not at all unusual—then these types of updates can reduce the average running times of searches, since frequently accessed values will generally be closer to the front of the list.

Use the rest of this page to present your implementation:

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

static bool contains(node *& list, int value) {
```

b. [5 points] Implement a function called **`mirror`** which accepts a linked list of integers and appends the reverse of that list to its end, resulting in a list that's twice the length. That means that **`mirror`** would transform the list

$$\mathbf{3 \rightarrow 4 \rightarrow 1 \rightarrow 5 \rightarrow 1}$$

into

$$\mathbf{3 \rightarrow 4 \rightarrow 1 \rightarrow 5 \rightarrow 1 \rightarrow 1 \rightarrow 5 \rightarrow 1 \rightarrow 4 \rightarrow 3}$$

Use the same node definition used for part a, and use the rest of this page to present your implementation:

```
static void mirror(node *list) {
```

Problem 2: Trie Insertion Trace [10 points]

Assume the following node definition for a trie:

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

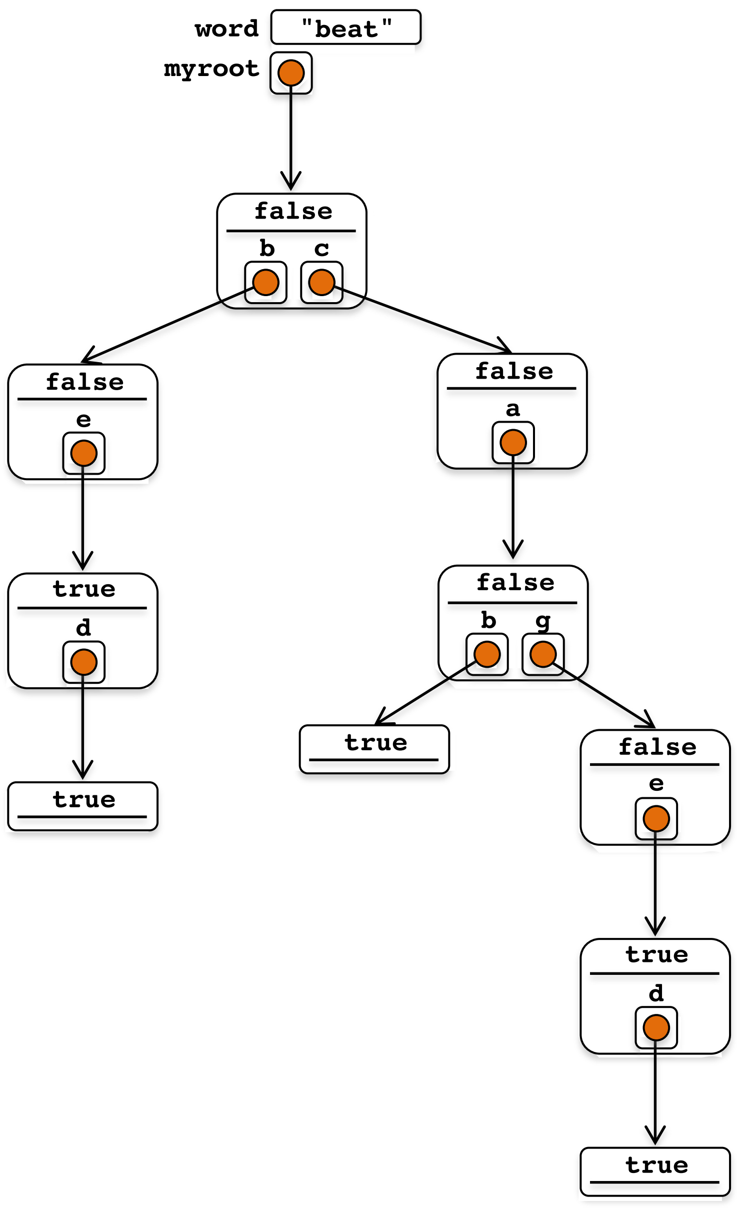
and consider the following two recursive implementations, each of which works to ensure the nodes needed to encode a word all exist.

```
node *ensureNodeExists1(node *root, const string& str, int pos = 0) {
   if (root == NULL) root = new node;
   if (pos == str.size()) return root;
   node *child = root->suffixes[str[pos]];
   return ensureNodeExists1(child, str, pos + 1);
}

node *ensureNodeExists2(node *& root, const string& str, int pos = 0) {
   if (root == NULL) root = new node;
   if (pos == str.size()) return root;
   node *&child = root->suffixes[str[pos]];
   return ensureNodeExists2(child, str, pos + 1);
}
```

For this problem, you're to assume the illustration to the right precisely captures how **`word`** (of type **`string`**) and **`myroot`** (of type **`node *`**) have been initialized just prior to a call one of the two functions above.

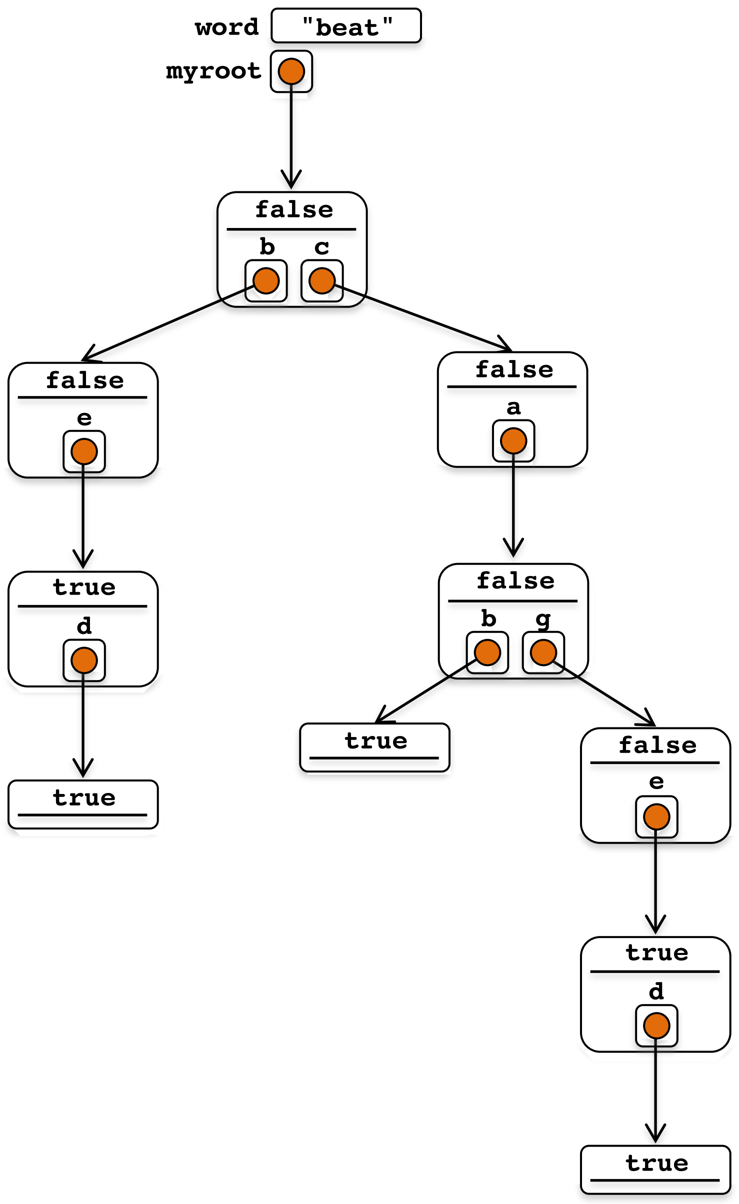
On the next two pages, you'll draw full memory traces to highlight why **`ensureNodeExists1`** fails to fully work even though **`ensureNodeExists2`** does.



Problem 2: Trie Insertion Trace [continued]

a. [5 points] Draw the state of memory when a call to **`ensureNodeExists1(myroot, word)`** bottoms out, just before its **`return root`** statement executes. You'll want to draw all of the parameters for all recursive calls, being clear what each of the parameters associated with each of the recursive calls contains. If you need to add key values to **`suffixes`** in any of the existing nodes, just draw them in as cleanly as possible.

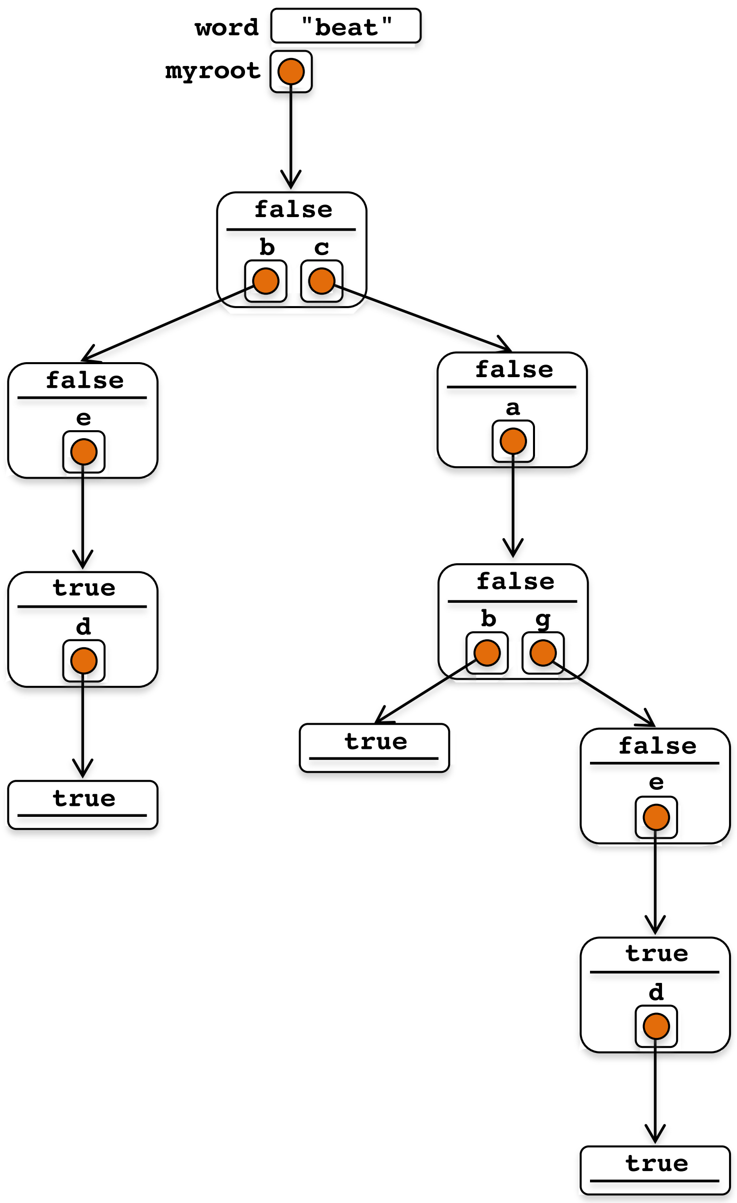
```
node *ensureNodeExists1(node *root, const string& str, int pos = 0) {
   if (root == NULL) root = new node;
   if (pos == str.size()) return root;
   node *child = root->suffixes[str[pos]];
   return ensureNodeExists1(child, str, pos + 1);
}
```



Problem 2: Trie Insertion Trace [continued]

b. [5 points] Now draw the state of memory when **`ensureNodeExists2(myroot, word)`** bottoms out, just before its own **`return root`** statement executes. You'll want to draw all of the parameters for all recursive calls, being clear what each of the parameters associated with each of the recursive calls contains. If you need to add key values to **`suffixes`** in any of the existing nodes, just draw them in as cleanly as possible.

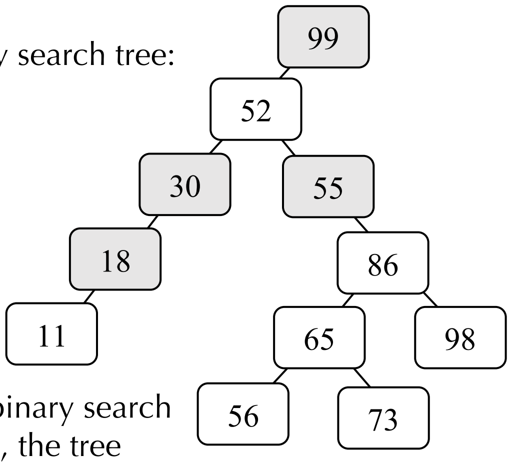
```
node *ensureNodeExists2(node *& root, const string& str, int pos = 0) {
   if (root == NULL) root = new node;
   if (pos == str.size()) return root;
   node *&child = root->suffixes[str[pos]];
   return ensureNodeExists2(child, str, pos + 1);
}
```



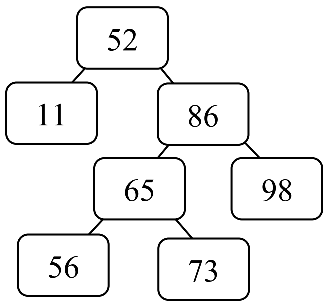
Problem 3: All Things Tree [15 points]

a. [7 points] Assume the following node definition for a binary search tree:

```
struct node {
   int value;
   node *left, *right;
};
```



Implement a function called **`contract`**, which converts a binary search tree into a full binary search tree by removing and deleting all half nodes—that is, internal nodes with only one child. If, for instance, the root of the binary search tree on the upper right is passed to our **`contract`** function, the tree would be transformed into the binary search tree below it.

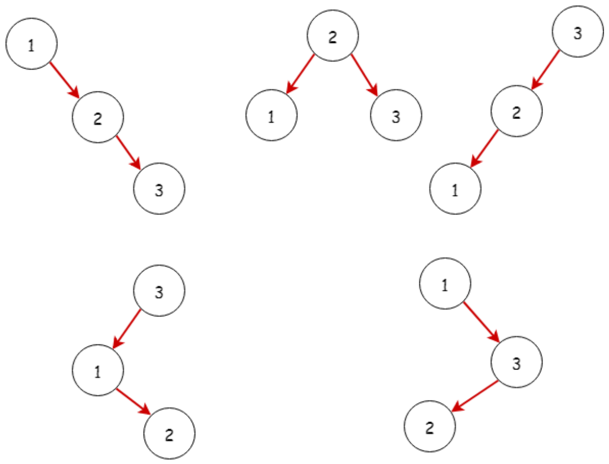


Use the rest of this page to present your recursive implementation:

```
static void contract(node *& root) {
```

Problem 3: All Things Tree [continued]

b. [8 points] There are five valid binary search tree structures on three keys, and you can see what they are on the right. For this problem, you're to write a function that recursively constructs all valid binary search storing the first n positive integers and returns them in an unordered **`Set<node *>`**. So, a call to **`construct(3)`** would return a **`Set`** storing the roots of the five trees on the right. None of the trees should share any memory whatsoever.



Use the rest of this page to present your implementation. You needn't worry about deleting any excess memory, and you can assume the existence of a **`cloneTree`** function, which accepts the root of a binary tree and returns the root of a deep copy—that is, a replica of the full tree that doesn't share any memory with the original.

```
static node *cloneTree(node *) { // assume it's implemented }
static Set<node *> construct(int n) {
```