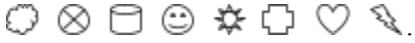
CS106B Practice Final Exam Solution

1. Binary Search Trees (BSTs)

(a) Valid, Valid, Invalid, Valid. *Note:* we can obtain a full accounting of their relationships by doing an in-order traversal of the tree, which gives the values in ascending order:
. That helps us quickly check the validity of trees.

(b) Worst, Worst, Not worst

2. Classes: Implementing Lexicon

(a) 8 (count how many "true" in the tree)

(b) `"a"` (the quotes aren't required, so plain a is fine, but `'a'` really isn't correct, since that's explicitly typed as `string` not `char`—we did not deduct for this though)

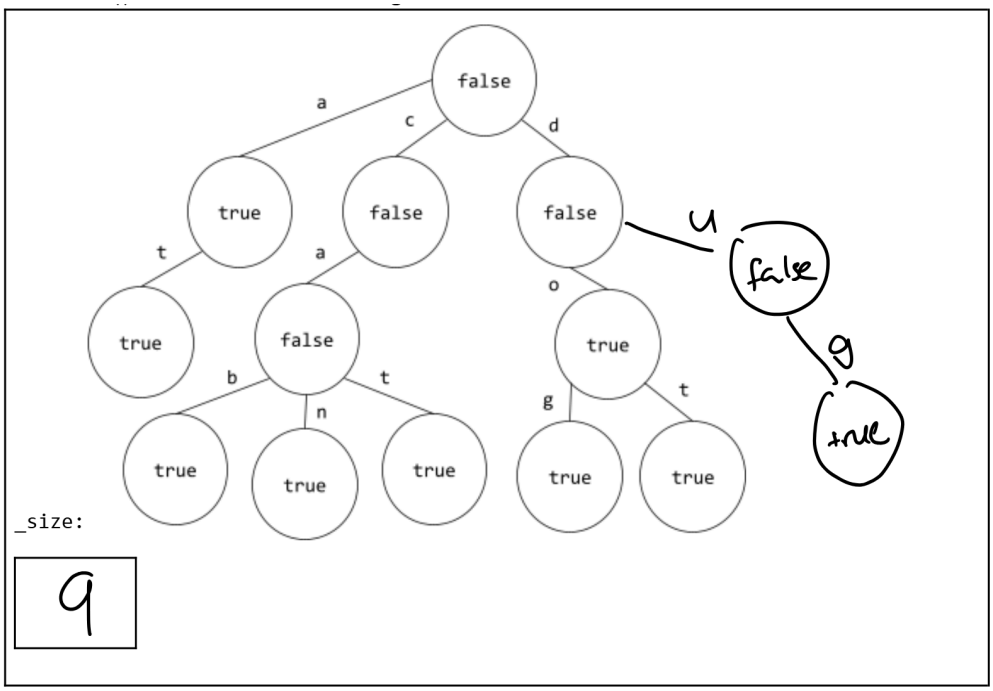
(c)

```C/C++
Lexicon::Lexicon() {
   _size = 0;
   _root = new Node;
   _root->isWordEnd = false;
}
```

An alternative solution uses struct initialization:

```C/C++
Lexicon::Lexicon() {
   _size = 0;
   _root = new Node {false, {}};
}
```

(d) Note that since `"cat"` is already an element of the set, we do not increase the `_size` member variable, or do any other modifications to the state, for that call to `add()`.



(e)

```C/C++
void Lexicon::add(string word) {
   Node* cur = _root;
   // iterate over characters to follow path
   for (char c : word) {
      // add new node on path if needed
      if (!cur->children.containsKey(c)) {
         cur->children[c] = new Node;
         cur->children[c]->isWordEnd = false;
      }
      cur = cur->children[c];
   }
   // only update if word is not already stored
   if (!cur->_isWordEnd) {
      cur->_isWordEnd = true;
      _size++;
   }
}
```

(f)

```C/C++
void destructorHelper::helper(Node* cur) {
    // recursive calls for each child
    for (char c : cur->children) {
        destructorHelper(cur->children[c]);
    }
    // post-order so we don't touch deleted memory
    delete cur;
}

Lexicon::~Lexicon() {
   destructorHelper(_root);
}
```

Fun fact: No nullptr checks whatsoever are needed here, because the empty tree has one `Node`, so `_root` can never be `nullptr`. And the `children` Maps don't store keys unless there's a real `Node` ptr value, so you don't have any `nullptr` risk there either. That said, extra `nullptr` checks will be harmless so we didn't deduct if students had them.

3. Classes: Pointers and Linked Lists.

```
bool contains(LinkNode* list, LinkNode* sub) {
    if (sub == nullptr) {
        return true;
    } else if (list == nullptr) {
        return false;
    }
    if (list->data == sub->data) {
        return contains(list->next, sub->next);
    } else {
        return contains(list->next, sub);
    }
}
```

4. Sorting

5 6 8 4 2 8 3 7 1
1 6 8 4 2 8 3 7 5
1 2 8 4 6 8 3 7 5
1 2 3 4 6 8 8 7 5
1 2 3 4 6 8 8 7 5
1 2 3 4 5 8 8 7 6

1 2 3 4 5 6 8 7 8
1 2 3 4 5 6 7 8 8
1 2 3 4 5 6 7 8 8
1 2 3 4 5 6 7 8 8

5. Big-O

(a) $O(2^N)$

(b) $O(N)$