CS106X Midterm Examination II

This is closed book, closed notes, closed reader, closed everything exam. If you're taking the exam remotely, you can telephone Jerry at 415-205-2242 while taking the exam to ask questions.

Good luck!

Section Leader: _____

Last Name: _____

First Name: _____

I accept the letter and spirit of the honor code. I promise to write more neatly than I ever have before in my entire life.

(signed) _____

		Score	Grader
1. Equivalence Classes	[10]	_____	_____
2. New Jersey Counties	[10]	_____	_____
3. Linked Lists and Sets	[10]	_____	_____
4. Tries and Removing Words	[8]	_____	_____
5. Patricia Tree Traversal	[7]	_____	_____
6. Short Answer	[5]	_____	_____
Total	**[50]**	_____	_____

Summary of Relevant Data Types

```
class string {
   bool empty() const;
   int size() const;
   int find(char ch) const;
   int find(char ch, int start) const;
   string substr(int start) const;
   string substr(int start, int length) const;
   char& operator[](int index);
   const char& operator[](int index) const;
};

class Vector {
   bool isEmpty() const;
   int size() const;
   void add(const Type& elem); // operator+= used similarly
   void insert(int pos, const Type& elem);
   void remove(int pos);
   Type& operator[](int pos);
};

class Grid {
   int numRows() const;
   int numCols() const;
   bool inBounds(int row, int col) const;
   Type get(int row, int col) const; // cascade of operator[] also works
   void set(int row, int col, const Type& elem);
};

class Stack {
   bool isEmpty() const;
   void push(const Type& elem);
   Type pop();
};

class Queue {
   bool isEmpty() const;
   void enqueue(const Type& elem);
   Type dequeue();
};

class Map {
   bool isEmpty() const;
   int size() const;
   void put(const Key& key, const Value& value);
   void remove(const Key& key);
   bool containsKey(const Key& key) const;
   Value get(const Key& key) const;
   Value& operator[](const Key& key);
};

class Set {
   bool isEmpty() const;
   int size() const;
   void add(const Type& elem); // operator+= also adds elements
   bool contains(const Type& elem) const;
};
```

Problem 1: Equivalence Classes [10 points]

Two positive integers **m** and **n** are said to be modulo-**d** equivalent if and only if
m % d == n % d—that is, both **m** and **n** produce the same remainder when divided by **d**.
When two integers are modulo-**d** equivalent, we say they fall into the same *equivalence class*.
Partitioning an array of positive integers into its modulo-**d** equivalence classes amounts to the
creation of **d** new arrays and distributing the integers across them. Those integers modulo-**d**
equivalent to **0** are placed in the 0th array, those modulo-**d** equivalent to **1** are placed in the
next array, and so forth.

As an illustration, consider the following array of positive integers:

4	7	11	1	3	10	2	16	5	15	8	6

The three modulo-3 classes of this array could be represented by this array of three arrays:
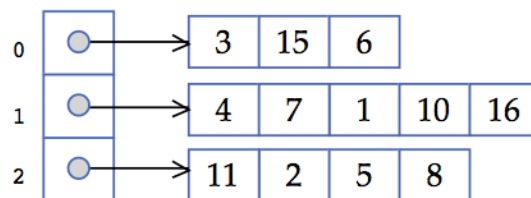
Note the all integers of the original array are represented exactly once, and that the perfect
multiples of 3 were placed in the 0th class, integers one more than a perfect multiple of 3 were
placed in the next class, and those two more than a perfect multiple of three were placed in
the final class. In general, an integer **n** is placed in the kth class whenever **n % 3** equals **k**.

The five modulo-5 classes of the original array would then be represented by this array of five
arrays:
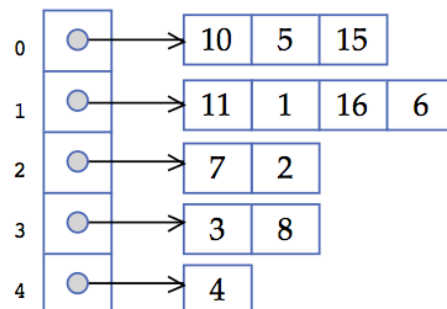
Again, the generalization here is that an integer **n** is placed in the kth array whenever **n % 5**
equals **k**.

a. [4 points] First, implement the **computeClassSizes** function, which takes an array, the array's size, and an integer **d**, and computes all of the sizes of the modulo-**d** classes. The function should dynamically allocate space for an integer array of length **d**, and should then go on to write all of the class sizes to that array. The size of the modulo-0 class should be stored at index **0**, the size of the modulo-1 class should be stored at index **1**, and so forth. You may assume that all integer arguments are positive, so that no error checking is necessary. The base address of the dynamically allocated array should be placed at the address supplied through the last parameter. You should not use any objects or records at all, as the intent here to exercise your understanding of primitives and raw memory. Note that most of the points are dedicated to the memory directives and syntax—operator **new**, **&**, *****, **->**, etc.—that you've learned in CS106X.

```
static void computeClassSizes(const int array[], int size,
                             int d, int **classSizes) {
```

b. [6 points] Now leverage your **computeClassSizes** function to implement the **partitionIntoClasses** function, which distributes the integers of the specified array over the **d** classes. The array of classes is dynamically allocated, and each class is itself dynamically allocated to store the exact number of integers that fall into that class. Ultimately, two arrays needs to be returned, so the base address of each should be stored at the two locations specified via **classes** and **classSizes**. Again, the proper use of **&**, *****, **[]**, and so forth really matter here, so be very careful with them. Don't use any objects and records.

```
static void partitionIntoClasses(const int array[], int size, int d,
                                 int ***classes, int **classSizes) {
```

Problem 2: New Jersey Counties [10 points]

Analyze the following code snippet, starting with a call to **somerset**, and draw the state of memory at the point indicated—just before the **bergen** helper function returns. 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. Generate your final diagram on the next page, and feel free to tear this page out so you can easily refer to it.

```
struct county {
    int atlantic;
    county *burlington[4];
    county **hudson;
};

static void somerset() {
    county salem[3];
    salem[0].atlantic = 8;
    salem[1].burlington[0] = NULL;
    salem[1].burlington[1] = salem;
    salem[1].burlington[2] = &salem[1];
    salem[1].burlington[3] = salem[1].burlington[1];
    county **ocean = &(salem->burlington[1]);
    ocean = &ocean[1];
    salem[1].hudson = ocean;
    ocean[1] = new county;
    ocean[1][0] = salem[1];
    bergen(salem[1], ocean, ocean);
}

static void bergen(county& passaic, county **warren, county **& sussex) {
    warren = new county *;
    *warren = &passaic;
    passaic.hudson = warren;
    sussex = passaic.burlington;
    *&sussex = NULL;
```
⟸ **Draw the state of memory just before bergen returns.**
```
}
```

**Problem 2: New Jersey County Pointer Trace [continued]**

Problem 3: Linked Links and Disjoint Sets [10 points]

Assume an unsorted set of distinct integers is stored in a linked list, where each node in the list stores not only the address of its successor, but also the address of the list's front node. The set itself tracks the addresses of both the head and tail nodes of the list, much like the **Queue** template container does.

For example, the set {4, 7, 8, 1} would look like this:
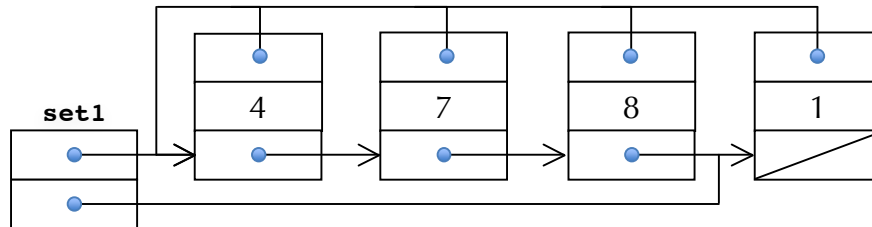
and the set {3, 9, 2} might look like this:
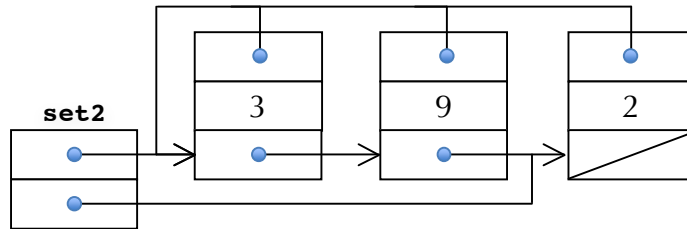
The data structures we'll use to support the above (we'll go with exposed **struct**s and not worry about encapsulation or object orientation) are as follows:

```
struct node {              struct set {
    node *front;               node *head;
    int value;                 node *tail;
    node *next;            };
};
```

One nifty feature of our linked list approach is that set unioning is conceptually trivial—particularly if we assume that the two sets being unioned are nonempty and have no elements in common. If, for instance, **set1** and **set2** as drawn above were unioned into a third set, that third set would look like this:
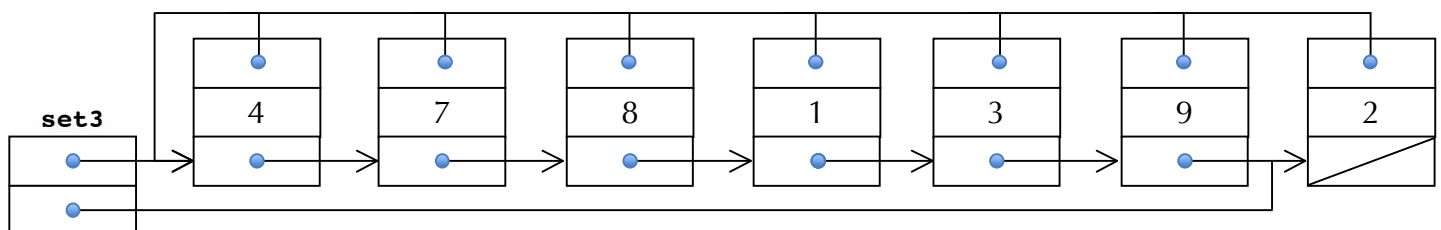
In particular, the tail node of the first set is updated to point to the head node of the second, and all **front** fields in the second set's list are updated to point to the front node of the first. No memory is allocated anywhere, as the nodes of the two original sets are donated to the third.

a. [5 points] Implement the **constructUnion** function, which accepts two sets by reference and returns their union (reusing existing nodes instead of allocating new ones). Assume the two incoming sets are nonempty and disjoint (e.g. no elements in common), and don't worry about cleaning up the two incoming sets in any way, as it's reasonable to assume that those two sets won't be used anymore, as their memory is being cannibalized and donated to a third one.

```
static set constructUnion(set& set1, set& set2) {
```

b. [5 points] Of course, one needs to be able to construct these sets in the first place. Implement the **buildSet** function, which accepts the provided **Vector<int>**— assumed to be nonempty and free of duplicates—and constructs and returns the equivalent set as described above. Recall that the linked list backing the set need not be sorted. You'll need to dynamically allocate nodes—one for each number in the referenced **Vector<int>**—and wire everything up as described above.

```
static set buildSet(Vector<int>& numbers) {
```

Problem 4: Tries and Removing Words [8 points]

Recall that the primary, object-oriented data structure we used to build a trie is defined as:

```
struct node {
   bool isWord;
   Map<char, node *> suffixes;
   node() { isWord = false; }
   ~node() { foreach (char ch in suffixes) delete suffixes[ch]; }
};
```

Implement the **remove** function, which removes the specified **word** from the trie, being careful to dispose of any nodes for prefixes that are no longer prefixes. To illustrate, let's inspect the sample trie I presented in lecture (presented on the right), which encodes five words: **"be"**, **"bed"**, **"cab"**, **"cage"**, and **"caged"**. If the word **"be"** is removed, then a single **true** if changed to **false** and that's that. No nodes are deleted, because all nodes relevant to **"be"** are needed to encode **"bed"**. If after removing **"be"** we remove **"bed"**, then we need to remove quite a few nodes, because **"b"**, **"be"**, and **"bed"** are no longer prefixes of anything. Additionally, the **'b'** entry in the root node's **suffixes** map would need to be removed as well.
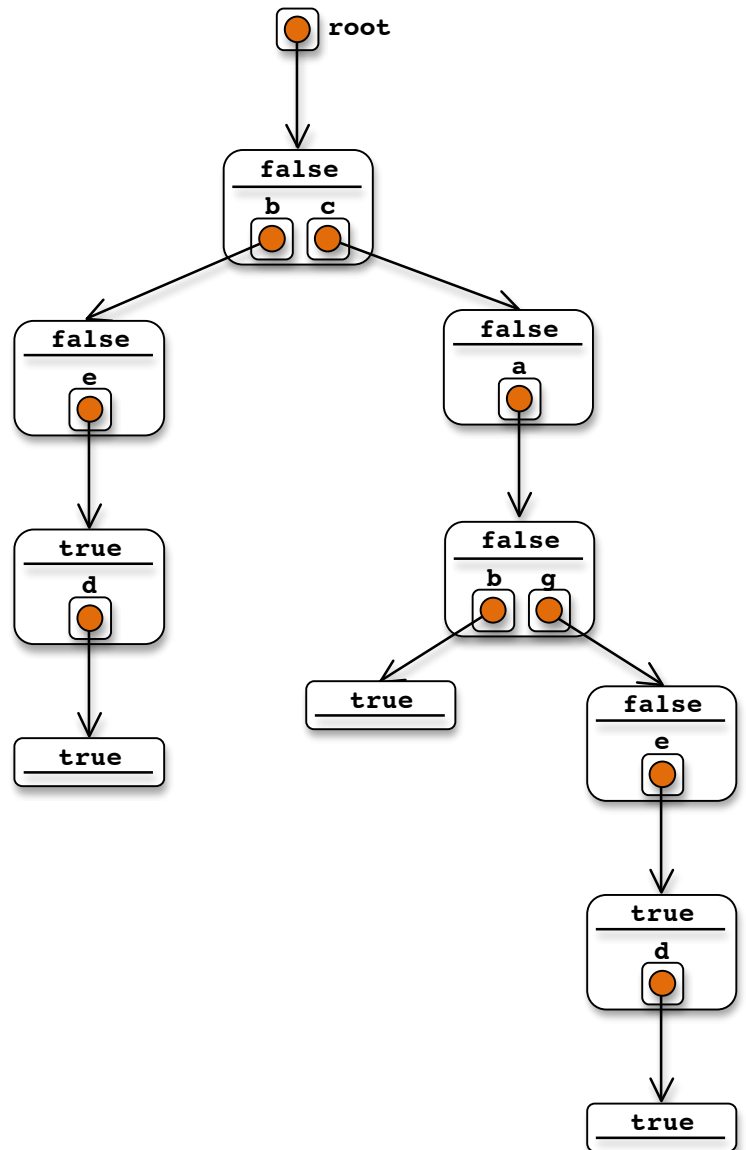
Special care must also be taken to see if the last remaining word is removed, in which case all remaining nodes would need to be killed off and the root would need to be set to **NULL**. Oh, and if the word isn't actually present, then you should just return without modifying the trie.

Using the next page, present your implementation of the **remove** function. Feel free to tear this page out so you can easily refer to it.

Problem 4: Tries and Removing Words [continue]

```
static void remove(node *& root, const string& word) {
```

Problem 5: Patricia Tree Traversal [7 points]

This problem leverages a data structure introduced in an earlier section handout, but the theory backing it is reproduced here in its entirety. Let's again consider the following illustration:
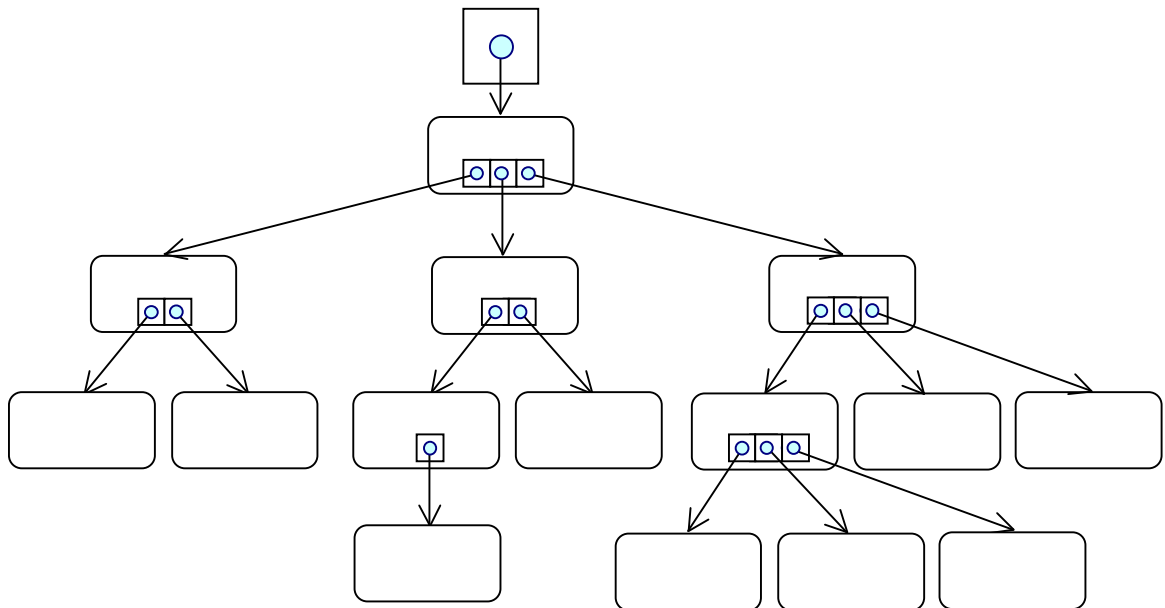
What's drawn above is an example of a **Patricia tree**—similar to a trie in that each node represents some prefix in a set of words. The child pointers, however, are more elaborate, in that they not only identify the sub-tree of interest, but they carry the substring of characters that should contribute to the running prefix along the way. Sibling pointers aren't allowed to carry substrings that have common prefixes, because the tree could 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 also 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*.

These two 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
};
```

Implement the **collectAllWords** function, which traverses the supplied tree as necessary and populates the referenced **Vector**—assumed to be empty starting out—so that by the end of execution it contains all of the tree's strings, sorted low to high. You should use recursion, and you'll want to implement this using a wrapper function.

```
static void collectAllWords(const node *root, Vector<string>& words) {
```

Problem 6: Short Answers [5 points]

a. [1 point] Assume some permutation of the numbers 1 through 3 are inserted into an initially empty binary search tree, and further assume that the resulting tree is as balanced as possible, even though the binary tree doesn't use any rotation or self-balancing strategies. How many different permutations of the numbers 1 through 3 result in a perfectly balanced tree?

b. [1 point] What would the Huffman encoding tree for the string **`bbbbaaacdcabbbb`** look like? (Don't worry about the pseudo-EOF, and understand that there are many correct encoding trees, not just one.)

c. [1 point] Describe two key differences between a reference and a pointer (e.g. contrast **`int&`** to **`int*`**).

d. [1 point] Explain why the **merge** operation for binomial-heap-backed priority queues is so much faster than the **merge** operations for the unsorted-vector-, the sorted-doubly-linked-list-, and the binary-heap-backed priority queues.

e. [1 point] Briefly describe two ways to reduce the memory footprint of the trie without compromising the execution time of **contains** and **containsPrefix**.