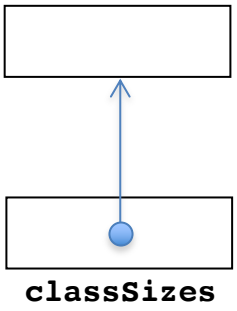
CS106X Practice Exam Solution

Solution 1: Equivalence Classes

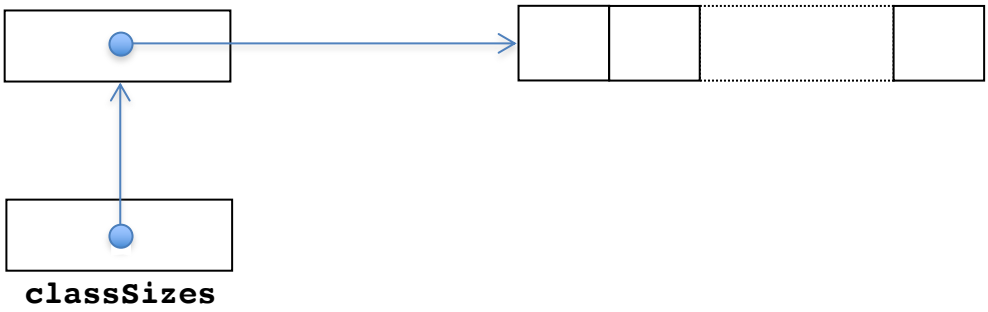
a. The problem was designed to be an algorithmically straightforward vehicle for pointer and memory jockeying. I present two solutions here, because we saw two different approaches to this problem when we graded the actual final that contained it.

```
static void computeClassSizes(const int array[], int size,
                              int d, int **classSizes) {
   *classSizes = new int[d];
   for (int i = 0; i < d; i++) (*classSizes)[i] = 0;
   for (int i = 0; i < size; i++) (*classSizes)[array[i] % d]++;
}
```

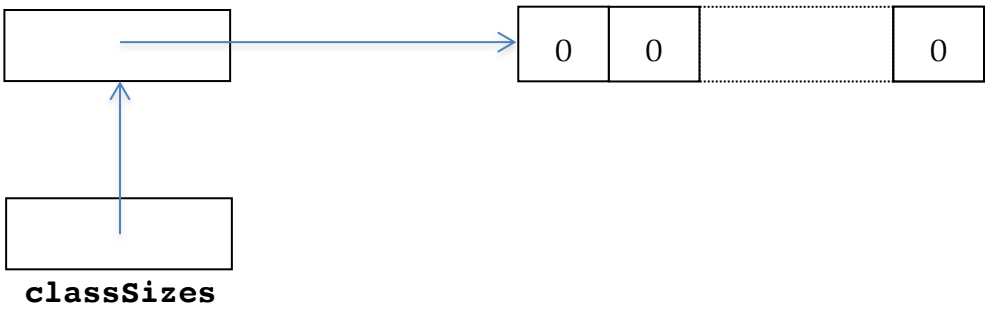
Presumably, the location of some **`int *`** has been shared via the **`classSizes`** parameter:



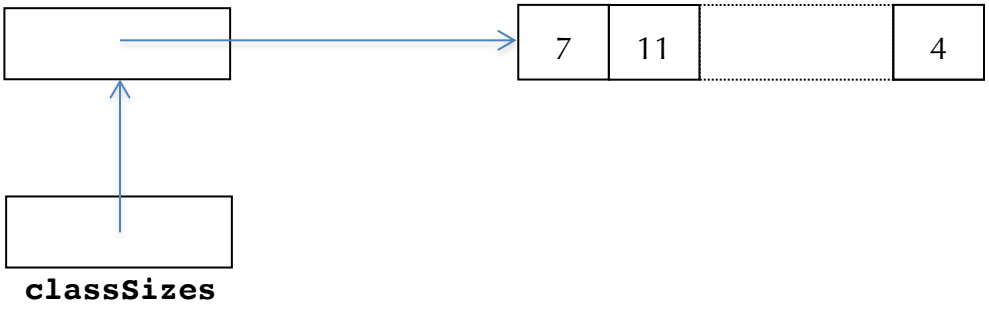
The first line of the three-line implementation is the tricky part. That line dynamically allocates a new array of integers, and places its base address in the space addressed by **`classSizes`**.



The second line dereferences **`classSizes`** to identify the base address of the array, and then further indexes to that base address to figure out where to place all of the zeroes.



The third line uses the same technique to identify one of several integers in the array so that a **++** can be levied against it. After the third line's **for** loop executes, the state of the array might look like this (I'm just making up some positive numbers that might result from all of the increments):



If asterisks make you sad, you can always allocate the array, place its address in the space addressed by **classSizes**, and then create a local copy of that base address and work through that. Here's a second version that's less intense on the asterisk front:

```
static void computeClassSizes(const int array[], int size,
                              int d, int **classSizes) {
   *classSizes = new int[d];
   int *sizesArray = *classSizes;
   for (int i = 0; i < d; i++) sizesArray[i] = 0;
   for (int i = 0; i < size; i++) sizesArray[array[i] % d]++;
}
```

Problem 1a Criteria: 4 points

- Makes proper call to **operator new**: 1 point
- Properly plants address of dynamically allocated figure in space addressed by **classSizes**: 1 point
- Understands the syntax needed to write to the dynamically allocated array entries: 1 point
- Properly updates the slots to contain the correct numbers (this covers the zeroing and **++**) business: 1 point

b. I'm sure you were all charmed by the triple pointer, but I need to drive home the fact that all pointers are, in a sense, *single* pointers that store the address of something else. In this case, **`classes`** stores the address where the base address of a dynamically allocated two-dimension array should be placed.

My solution is presented here:

```
static void partitionIntoClasses(const int array[], int size, int d,
                                 int ***classes, int **classSizes) {
   computeClassSizes(array, size, d, classSizes);
   *classes = new int *[d];
   for (int i = 0; i < d; i++) (*classes)[i] = new int[(*classSizes)[i]];
   for (int i = 0; i < d; i++) (*classSizes)[i] = 0;
   for (int i = 0; i < size; i++) {
      (*classes)[array[i] % d][(*classSizes)[array[i] % d]] = array[i];
      (*classSizes)[array[i] % d]++;
   }
}
```

Here are some key observations to defend each line of my solution:

- The first line passes the buck to the previously written **`computeClassSizes`** function. Conceptually it makes sense, I'm sure, but the hard part was understanding what to pass through via the fourth argument. **`partitionIntoClasses`** receives the location where the base address of a sizes array should be placed, and that location needs to shared with **`computeClassSizes`**. Many of you might have been tempted to pass through **`&classSizes`**, but that would be passing an **`int ***`** where an **`int **`** is expected.
- The second line dynamically allocates the spine of a two-dimensional integer array and places it in the space shared via the **`classes`** parameter.
- The third line walks over the vertebrae of the spine and loads each one with the base address of another array just big enough to store all of the numbers that fall into the corresponding equivalence class.
- The fourth line—the second **`for`** loop—resets all of the slots in the **`(*classSizes)`** array to be 0 so they can operate as index variables into the equivalence class arrays.
- The final **`for`** loop distributes each of the numbers in **`array`** to the correct equivalence class, and in the process advances each value of **`(*classSizes)[i]`** back up to the value it held before we zeroed them all out.

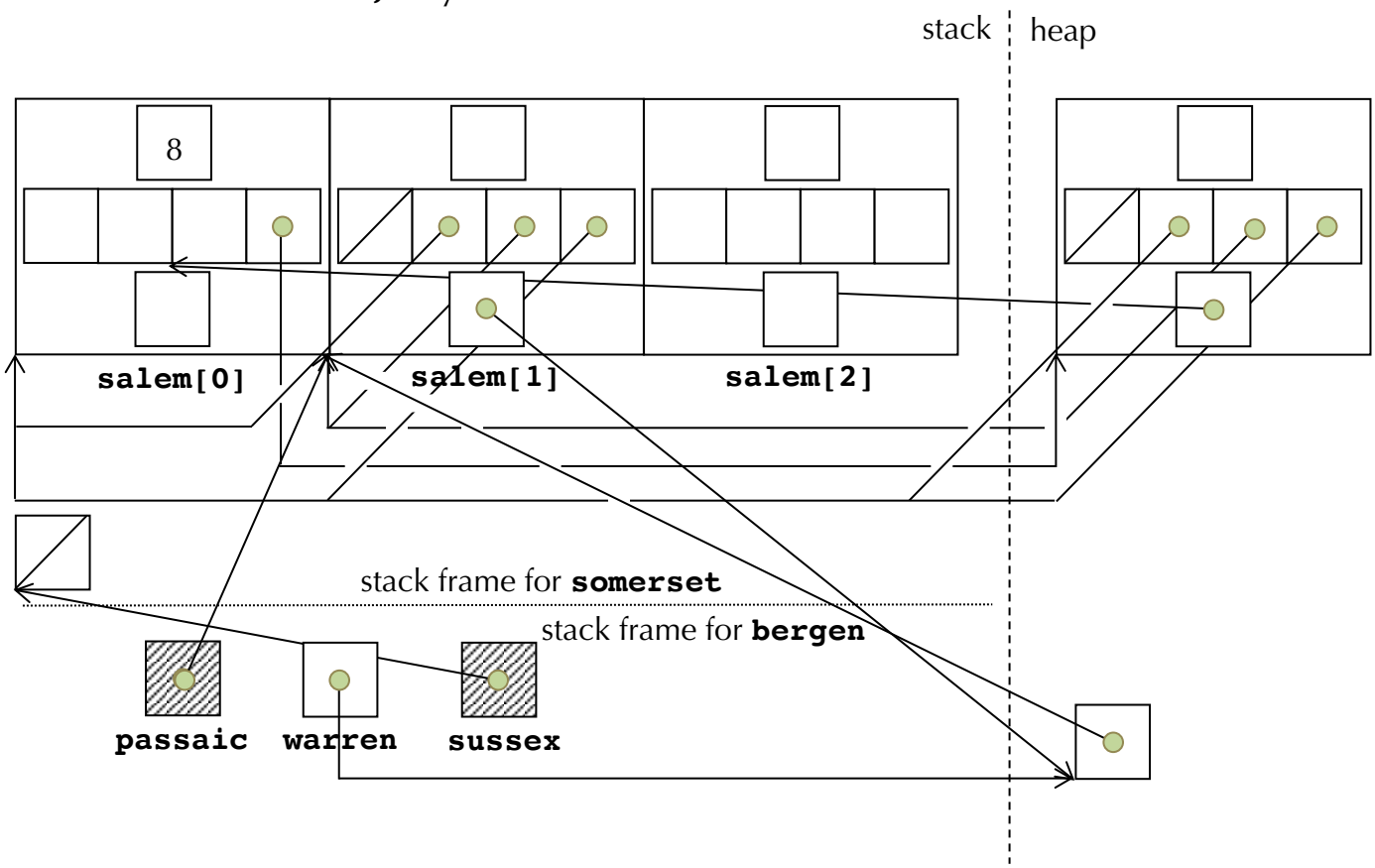
If the repeated application of **`operator*`** over and over again seems academic, you can go with a similar approach to that used by my second solution to part a.

```
static void partitionIntoClassez(const int array[], int size, int d,
                                 int ***classes, int **classSizes) {
   computeClassSizes(array, size, d, classSizes);
   *classes = new int *[d];
   int *sizesArray = *classSizes;
   int **classesArray = *classes;
   for (int i = 0; i < d; i++) classesArray[i] = new int[sizesArray[i]];
   for (int i = 0; i < d; i++) classesArray[i] = 0;
   for (int i = 0; i < size; i++) {
      classesArray[array[i] % d][sizesArray[array[i] % d]] = array[i];
      sizesArray[array[i] % d]++;
   }
}
```

Problem 1b Criteria: 6 points

- Properly calls **`computeClassSizes`** with the correct first and fourth arguments (**`array`**, not **`&array`** or **`*array`**, and **`classSizes`**, not **`*classSizes`** or **`&classSizes`**—just **`classSizes`**): 2 points, 1 point for each
- Properly allocates the spine of the two-dimensional array and places it in the space addressed by **`classes`**: 1 point
- Properly allocates the equivalence class arrays and places their base addresses within the spine: 1 point
- Presents an acceptable algorithm to distribute integers across the two-dimensional array: 2 points
    - Approach is algorithmically sound: 1 point
    - Approach is properly implemented: 1 point

Solution 2: New Jersey Counties



Problem 2 Criteria: 10 points

The rubric is simple: Take off 0.5 points for each error, save for the following:

- If the array of length 3 is drawn with separations in between the records, just comment that they are contiguously laid down, but don't take off any points.
- The accumulation of all side effects associated with the line **`ocean[1][0]`** = **`salem[1]`** is worth 1 point.
- The initialization of **`bergen`**'s first and third parameters are worth 0.5 points each (the second parameter is immediately reassigned, so it's impossible to see if it was initialized properly. Just ignore it.)
- Each error with orphaned memory (e.g. they say memory is orphaned when it isn't) is worth 0.5 points. There are only two dynamically allocated figures that could be identified as orphaned, but just in case, cap deductions related to orphaned memory at 1 point
- If you can't tell the stack from the heap, take off 0.5 points.

Solution 3: Linked Links and Disjoint Sets

a. I promised the two sets would be nonempty just to reduce the number of special cases. In order to merge two sets, we need to:

- update every node in the second set to point to the leading node of the first (the first **for** loop of my solution below takes care of this),
- update the non-**NULL** tail of **set1** to point to the leading node of **set2** (handled by the third to last line of my solution) and set the tail of **set1** to be the tail of **set2**.
- return the merged set (and because we know **set1** and **set2** are being cannibalized, we can fold everything into **set1** and return it as the unioned set)

```
static set constructUnion(set& set1, set& set2) {
   for (node *curr = set2.head; curr != NULL; curr = curr->next) {
      curr->front = set1.head;
   }

   set1.tail->next = set2.head;
   set1.tail = set2.tail;
   return set1;
}
```

Problem 3a Criteria: 5 points

- Properly uses **.** to access fields of items referenced by **set1** and **set2**, and **operator->** to access fields of figures addressed by **node** ***s: 1 point
- Properly iterates over all nodes of second set and updates the **front** pointers to address head of first: 1 point (from this point forward on this problem, forgive mix-ups regarding **.**, ***, and **->**, since all mistakes will just be the same. This point is dedicated to the proper **for** loop idiom and accessing **front** as an l-value and **head** as an
  r-value.)
- Properly updates original tail node of first set to point to **head** of second: 1 point
- Properly updates tail of the first to be tail of the second: 1 point
- Properly bundles the union into a new **set**, or into **set1**, and returns it without regard for destruction of **set1** and/or **set2**: 1 point

b. This second part was intended to exercise your ability to synthesize a linked list from scratch using **operator new**.

```
static node *createNode(node *front, int value, node *next) {
   node n = {front, value, next};
   return new node(n);
}

static set buildSet(Vector<int>& numbers) {
   set s;
   s.head->front = s.head = s.tail = createNode(NULL, numbers[0], NULL);
   for (int i = 1; i < numbers.size(); i++) {
      s.tail->next = createNode(s.head, numbers[i], NULL);
      s.tail = s.tail->next;
   }
   return s;
}
```

We're allowed to assume the incoming **Vector** is nonempty. And because every node needs to point back to the first, I need to ensure that the leading node exists before I create all others (hence the one-off call to **createNode** before the **for** loop). The other numbers can be inserted either front to back (as I did above) or back to front, but care needs to be taken to make sure the set's **tail** pointer ends up pointing to the very last node. Some of you iterated from back to front, which is of course fine as well. Here's a nice compact way of doing that:

```
static set buildSet(Vector<int>& numbers) {
   set s;
   s.head->front = s.head = s.tail = createNode(NULL, numbers[0], NULL);
   for (int i = numbers.size() - 1; i > 1; i--) {
      s.head->next = createNode(s.head, numbers[i], s.head->next);
      if (i == numbers.size() - 1) s.tail = s.tail->next;
   }
   return s;
}
```

Ashwin Siripurapu, a section leader during the quarter I gave this final exam, thought some of you might take a double-pointer approach, and provided this alternate solution:

```
static set buildSet(Vector<int>& numbers) {
   node *head = new node;
   head->value = numbers[0];
   node *tail = head->front = head;
   node **pnext = &(head->next);
   for (int i = 1; i < numbers.size(); i++) {
      node n = {head, numbers[i], NULL};
      tail = *pnext = new node(n);
      pnext = &(tail->next);
   }
   *pnext = NULL;
   set s = {head, tail};
   return s;
}
```

Problem 3b Criteria: 5 points

Understand that the **.** and **->** business around static **set**s and **node ***s can't be the subject of point loss for part b, since I don't want to double-ding for the same error within the same problem over and over.

- Correctly manages to allocate nodes for all numbers: 1 point
- Properly wires up all **next** pointers to address their successor (or **NULL**): 1 point
- Properly wires up all **front** pointers, including that within the leading node, to point to the leading node: 1 point (this might just be done in a second pass over the list, which is fine)
- Properly sets **head** and **tail** fields of a set and returns it: 1 point

Properly manages memory (no orphaned memory), and no new types of syntax errors with **&** or *, etc.: 1 point

Solution 4: Tries and Removing Words

This was the most demanding problem on the entire exam. To make it easier, you were to assume the trie was always well formed in that there are no **NULL** pointer values in the **Map**s, and that all of the leaf nodes in the trie have their **isWord** fields set to **true**.

```
static bool deleteIfChildless(node *& root) {
   if (!root->isWord && root->suffixes.isEmpty()) {
      delete root;
      root = NULL;
   }
   return root == NULL;
}

static bool removeSuffix(node *& root, const string& suffix) {
   if (suffix.empty()) {
      root->isWord = false;
   } else {
      char ch = suffix[0];
      if (!root->suffixes.containsKey(ch) ||
          !removeSuffix(root->suffixes[ch], suffix.substr(1))) return false;
      root->suffixes.remove(ch);
   }

   return deleteIfChildless(root);
}

static void remove(node *& root, const string& word) {
   if (root != NULL) {
      removeSuffix(root, word);
   }
}
```

Problem 4 Criteria: 8 points

- Using either recursion (via **node *&** parameters) or iteration (via **node **** variables), properly discovers the node that represents the word being removed (if it exists): 2 points
    - Properly bottoms out when all characters have been accounted for without crashes or any serious memory flaws: 1 point

    - Properly bottoms out when the next character isn't encoded in the **`suffixes Map`**: 1 point
- If the node doesn't exist, then ultimately returns without modifying tree: 1 point
- If the node exists but it represents a prefix but not a word, then returns without modifying the tree: 1 point
- If the node represents a word, then updates **`isWord`** to be **`false`**: 1 point
- If the node represents a word that isn't a prefix of larger words, correctly codifies the removal of all extraneous nodes by de-allocating them and removing their footprints from the **`Map`** in the parent: 2 points (points awarded via bucket system)
    - Completing nails everything about it or misses something trivial that'd be detected in a real coding environment): 2 points
    - Codifies the de-allocation, but fails to properly communicate that information to the parent so it can remove the entry that led to the now-dead node: 1 point
    - Codifies the de-allocation and the upward communication, but accidentally deletes parent nodes even when they represent words that are still part of the trie: 1 point
    - The back-chaining de-allocation effort is missing or woefully incorrect: 0 points

Solution 5: Patricia Tree Traversal

```
static void collectAllWords(const node *root,
                            const string& prefix, Vector<string>& words) {
   if (root->isWord) words += prefix;
   foreach (const connection& conn: root->children) {
      collectAllWords(conn.subtree, prefix + conn.letters, words);
   }
}

static void collectAllWords(const node *root, Vector<string>& words) {
   if (root == NULL) return; // subtree fields never NULL, but root might be
   collectAllWords(root, "", words);
}
```

Each node in the Patricia tree represents some prefix, and that prefix is the concatenation of the letters within the **`connection`**s that led to it. If a node's **`isWord`** variable is **`true`**, then we know the running prefix is actually a word, and that it should be appended to the series of words we're building up in the referenced **`Vector`**. Regardless of **`isWord`**'s value, we should recursively descend through each of the node's **`connection`**s, extending the running prefix as appropriate.

Problem 5 Criteria: 7 points

- Properly guards against the empty tree: 1 point (either before the recursion begins, or because they have a **`NULL`** check at the top of their recursive implementation)
- Properly frames the original in terms of a wrapper that actually manages the recursion and supplies the **`""`** to start: 1 point
- Properly examines the **`isWord`** field within the **`root`**, and prints the running prefix if it's **`true`**: 1 point

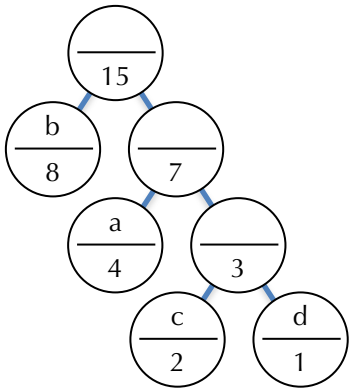
- Does the above without returning (for a word may be a prefix of a longer word): 1 point
- Uses the correct idiom (**`foreach`**, **`for`** loop, **`while`** loop) to iterate over all **`connection`**s descending from the root (**`const connection&`** isn't necessary—data type of **`connection`** is fine): 1 point
- Makes proper use of **`*`**, **`.`**, **`->`**, and/or **`[]`** as appropriate to access **`connection`** fields and pass through new arguments to recursive calls: 1 point
- Correctly passes the correct sub-tree and the corresponding prefix to the recursive call: 1 point

Solution 6: Short Answers

The criteria should be self-explanatory. Where only one answer is expected, either 0 or 1 point should be given. Where two answers are expected, give 0.5 points for one correct response paired with one incorrect, mostly redundant, or completely missing response. If they provide three answers where two are expected, only grade the first two and ignore others.

a. There are six insertion orders, and just *two* of them result in a balanced tree. 2 must be inserted first, and then 1 and 3 can be inserted in either order.

b. One of 8 possible answers:



c. There are several differences, and we're happy to give credit for any two that didn't say the same thing.

- References are automatically dereferenced, whereas pointers are not and must be programmatically dereferenced via **`operator*`**.
- Pointers themselves can be assigned to point to different memory locations during the course of execution, whereas references are bound to refer to the same location for their entire lifetime. Along the same lines, pointer variables need not be initialized immediately, but reference variables must be associated with some figure in memory on the same line they're declared (e.g. uninitialized pointers compile, but uninitialized [or dangling] references do not).
- One must use **`operator&`** to pass the address of a figure in memory, whereas **`operator&`** is not needed to pass a figure by reference.

d. Merging unsorted vectors, sorted doubly-linked lists, and array-backed binary heaps requires all elements in one or both be touched, resulting in an execution time that's proportional to the number of elements in the smaller priority queue. Merging two structurally identical binomial heaps—regardless of their size—requires the examination of just two elements and a constant number of pointer updates. Merging two priority queues backed by arrays of binomial heaps takes time proportional to the logarithm of the size of the larger heap, which is sub-linear.

e. The correct answer was all assignments are your favorite. Hope you know that.