CS106X
Autumn 2012

Handout 37S
November 16th, 2012

CS106X Practice Exam Solution

Solution 1: Linked Lists

a.

```
static Map<string, string> concatenateMaps(const node *maplist) {
   Set<string> keys;
   for (const node *curr = maplist; curr != NULL; curr = curr->next) {
      foreach (string key in curr->map) {
         keys.add(key);
      }
   }

   Map<string, string> concatenations;
   foreach (string key in keys) {
      string value;
      for (const node *curr = maplist; curr != NULL; curr = curr->next) {
         if (curr->map.containsKey(key)) {
            value += curr->map.get(key);
         }
      }

      concatenations.put(key, value);
   }

   return concatenations;
}
```

b.

```
static void stretchList(node *list) {
   int copyCount = 1;

   while (list != NULL) {
      for (int i = 0; i < copyCount; i++) {
         node *newNode = new node;
         newNode->value = list->value;
         newNode->next = list->next;
         list->next = newNode;
         list = newNode;
      }
      list = list->next;
      copyCount++;
   }
}
```

c.

```
static void buildSeriallzationArray(const node *list, Vector<int>& values) {
   if (list == NULL) return;
   values.add(list->value);
   buildSeriallzationArray(list->down, values);
   buildSeriallzationArray(list->next, values);
}

static node *arrayToList(const Vector<int>& values) {
   node *head = NULL;
   for (int i = values.size() - 1; i >= 0; i--) {
      node *newNode = new node;
      newNode->value = values.get(i);
      newNode->down = NULL;
      newNode->next = head;
      head = newNode;
   }

   return head;
}

static node *flattenList(const node *list) {
   Vector<int> values;
   buildSeriallzationArray(list, values);
   return arrayToList(values);
}
```

d.

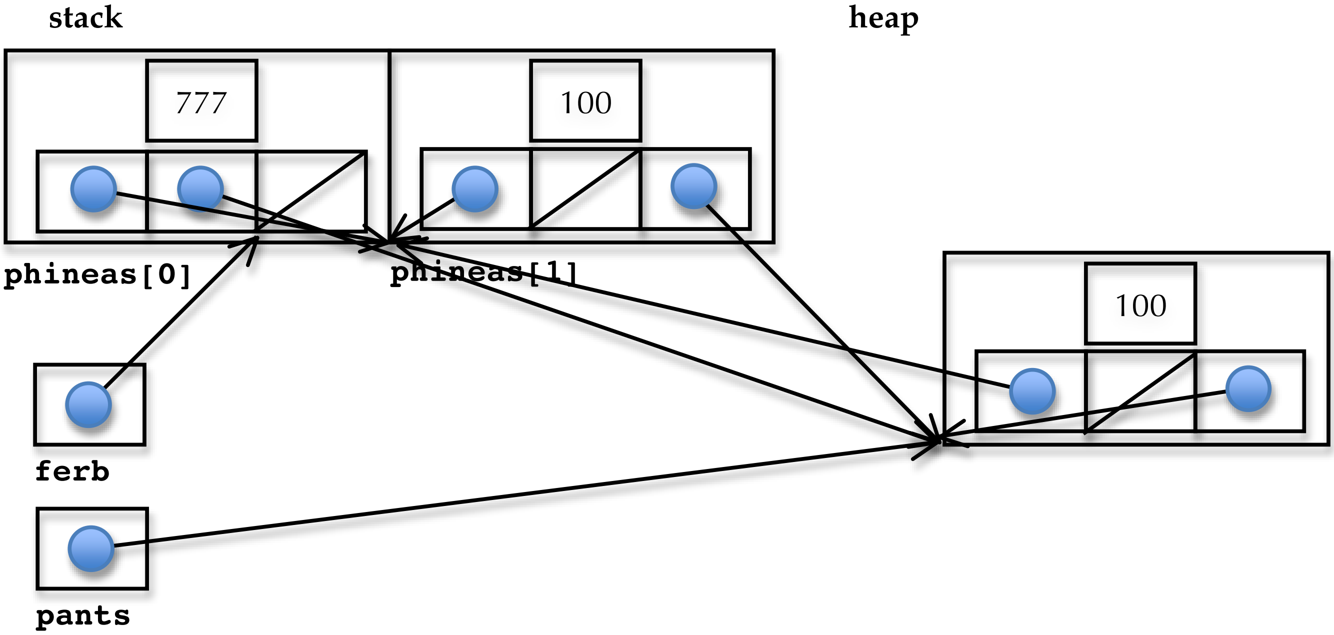
```
static node *generateSternBrocotTree(int denominator,
                                     const fraction& low, const fraction& high) {
   fraction mediant = {
      low.numerator + high.numerator,
      low.denominator + high.denominator
   };

   if (mediant.denominator > denominator) return NULL;

   node *root = new node;
   root->value = mediant;
   root->left = generateSternBrocotTree(denominator, low, mediant);
   root->right = generateSternBrocotTree(denominator, mediant, high);
   return root;
}

static node *generateSternBrocotTree(int denominator) {
   fraction zero = {0, 1};
   fraction one = {1, 1};
   return generateSternBrocotTree(denominator, zero, one);
}
```

Solution 2: Phineas and Ferb



Solution 3: Encoding General Trees

```
static binTreeNode *encode(Vector<genTreeNode *>& siblings, int start) {
   if (start == siblings.size()) return NULL;
   binTreeNode *root = new binTreeNode;
   root->value = siblings[start]->value;
   root->left = encode(siblings[start]->children, 0);
   root->right = encode(siblings, start + 1);
   return root;
}

static binTreeNode *encode(genTreeNode *root) {
   Vector<genTreeNode *> rootAsVector;
   rootAsVector.add(root);
   return encode(rootAsVector, 0);
}
```

Solution 4: Dictionaries and Ternary Search Trees

a.

```
Dictionary::node *Dictionary::createNode(char ch) const {
   node n = {ch, NULL, NULL, NULL, NULL};
   return new node(n);
}

void Dictionary::add(const string& word, const string& definition) {
   if (word.empty()) error("The empty string cannot be entered");
   node **currp = &root;
   int len = word.size(), pos = 0;
   while (true) {
      if (*currp == NULL) *currp = createNode(word[pos]);
      node *curr = *currp;
      if (curr->letter == word[pos]) {
         pos++;
         if (pos == len) break;
         currp = &curr->equal;
      } else if (curr->letter < word[pos]) {
         currp = &curr->greater;
      } else {
         currp = &curr->less;
      }
   }
   if ((*currp)->definitions == NULL) (*currp)->definitions = new Vector<string>;
   (*currp)->definitions->add(definition);
}
```

b.

```
void Dictionary::deleteNode(node *curr) {
   if (curr == NULL) return;
   delete curr->definitions; // delete NULL is a no-op
   deleteNode(curr->less);
   deleteNode(curr->equal);
   deleteNode(curr->greater);
   delete curr;
}

Dictionary::~Dictionary() {
   deleteNode(root);
}
```