CS106B Handout 37S
Autumn 2012 November 28th – 30th, 2012

Section Solution

Discussion Problem 1 Solution: Quadtrees

```
static quadtree *gridToQuadtree(Grid<bool>& image,
                                int lowx, int highx, int lowy, int highy) {
   quadtree *qt = new quadtree;
   qt->lowx = lowx; qt->highx = highx - 1;
   qt->lowy = lowy; qt->highy = highy - 1;
   if (allPixelsAreTheSameColor(image, lowx, highx, lowy, highy)) {
      qt->isBlack = image[lowx][lowy];
      for (int i = 0; i < 4; i++) {
         qt->children[i] = NULL;
      }
   } else {
      int midx = (highx - lowx) / 2;
      int midy = (highy - lowy) / 2;
      qt->children[NW] = gridToQuadtree(image, lowx, midx, midy, highy);
      qt->children[NE] = gridToQuadtree(image, midx, highx, midy, highy);
      qt->children[SE] = gridToQuadtree(image, midx, highx, lowy, midy);
      qt->children[SW] = gridToQuadtree(image, lowx, midx, lowy, midy);
      // assume NW, NE, etc are constants/#defines
   }

   return qt;
}

static quadtree *gridToQuadtree(Grid<bool>& image) {
   return gridToQuadtree(image, 0, image.numCols(), 0, image.numRows());
}
```

I don't bother with the code for **`allPixelsAreTheSameColor`**, because it's all kinds of obvious.

Discussion Problem 2 Solution: Patricia Trees

The **`containsWord`** routine is nontrivial, because it's as much about trees as it is about advanced string manipulation. It's complicated by the fact that the letters in the **`connection`** may be longer than the remaining portion of the word.

```
static int findConnection(const Vector<connection>& children,
                          const string& word) {
   for (int i = 0; i < children.size(); i++) {
      string prefix = word.substr(0, children.get(i).letters.size());
      int cmp = prefix.compare(children.get(i).letters);
      if (cmp == 0) return i;
      if (cmp < 0) break;
   }

   return -1;
}
```

```
static bool containsWord(const node *root, const string& word) {
   const node *curr = root;
   string clone = word;
   while (!clone.empty()) {
      int index = findConnection(curr->children, clone);
      if (index == -1) return false;
      clone = clone.substr(curr->children.get(index).letters.size());
      curr = curr->children.get(index).subtree;
   }

   return curr->isWord;
}
```

Discussion Problem 3 Solution: Regular Expressions

```
static void matchAllWords(const node *root, const string& regex,
                          Set<string>& matches, const string& workingPrefix) {

   if (root == NULL) return;
   if (regex.empty()) {
      if (root->isWord) {
         matches.add(workingPrefix);
      }
      return; // return regardless of whether the working prefix was a word
   }

   if (regex.size() == 1 || !ispunct(regex[1])) {
      matchAllWords(root->suffixes.get(regex[0]), regex.substr(1),
                    matches, workingPrefix + regex[0]);
   } else if (regex[1] == '?') {
      matchAllWords(root, regex.substr(2), matches, workingPrefix);
      matchAllWords(root->suffixes.get(regex[0]), regex.substr(2),
                    matches, workingPrefix + regex[0]);
   } else if (regex[1] == '*') {
      matchAllWords(root, regex.substr(2), matches, workingPrefix);
      matchAllWords(root->suffixes.get(regex[0]), regex,
                    matches, workingPrefix + regex[0]);
   } else { // assume regex[1] == '+'
      string equivregex = regex;
      equivregex[1] = '*'; // reframe y+ as yy*
      matchAllWords(root->suffixes.get(equivregex[0]), equivregex,
                    matches, workingPrefix + equivregex[0]);
   }
}

static void matchAllWords(const node *trie, const string& regex,
                          Set<string>& matches) {
   matchAllWords(trie, regex, matches, "");
}
```

Lab Problem 1 Solution: Exponential Trees

The **`add`** operation is certainly the more challenging of the two methods, which is why it's the more interesting one to implement. I'm providing an implementation that makes use of double pointers, since many of you have confessed your fear of the double pointer and want more practice. Note I make use of a helper method called **`find`**, which would need to be mentioned in the **`private`** section of the **`class`** definition.

```
int ExponentialTree::find(Vector<string>& values, const string& value) const {
   for (int i = 0; i < values.size(); i++) {
      if (value < values[i]) {
         return i;
      }
   }

   return values.size();
}

void ExponentialTree::add(const string& value) {
   int depth = 1;
   node **currp = &root;
   while (*currp != NULL) {
      node *curr = *currp;
      int insertionPoint = find(curr->values, value);
      if (curr->values.size() < curr->depth) {
         curr->values.insert(insertionPoint, value);
         return; // we had space for it inside this node, so splice it in
      }

      if (curr->children == NULL) {
         curr->children = new node *[depth + 1];
         for (int i = 0; i <= depth; i++) {
            curr->children[i] = NULL;
         }
      }

      depth++;
      currp = &curr->children[insertionPoint];
   }

   node *curr = new node;
   curr->depth = depth;
   curr->values.add(value);
   curr->children = NULL;
   *currp = curr;
}
```