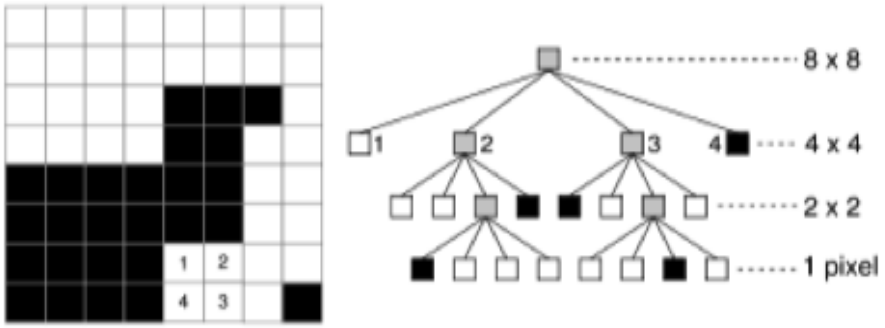
Section Handout

Discussion Problem 1: Quadtrees

A quadtree is a rooted tree structure where each internal node has precisely four children. Every node in the tree represents a square, and if a node has children, each encodes one of that square's four quadrants.

Quadtrees have many applications in computer graphics, because they can be used as in-memory models of images. That they can be used as in-memory versions of black and white images is easily demonstrated via the following (borrowed from Wikipedia.org):



The 8 by 8 pixel image on the left is modeled by the quadtree on the right. Note that all leaf nodes are either black or white, and all internal nodes are shaded gray. The internal nodes are gray to reflect the fact that they contain both black **and** white pixels. When the pixels covered by a particular node are all the same color, the color is stored in the form of a Boolean and all four children are set to **`NULL`**. Otherwise, the node's sub-region is recursively subdivided into four sub-quadrants, each represented by one of four children.

Given a **`Grid<bool>`** representation of a black and white image, implement the **`gridToQuadtree`** function, which reads the image data, constructs the corresponding quadtree, and returns its root. Frame your implementation around the following data structure:

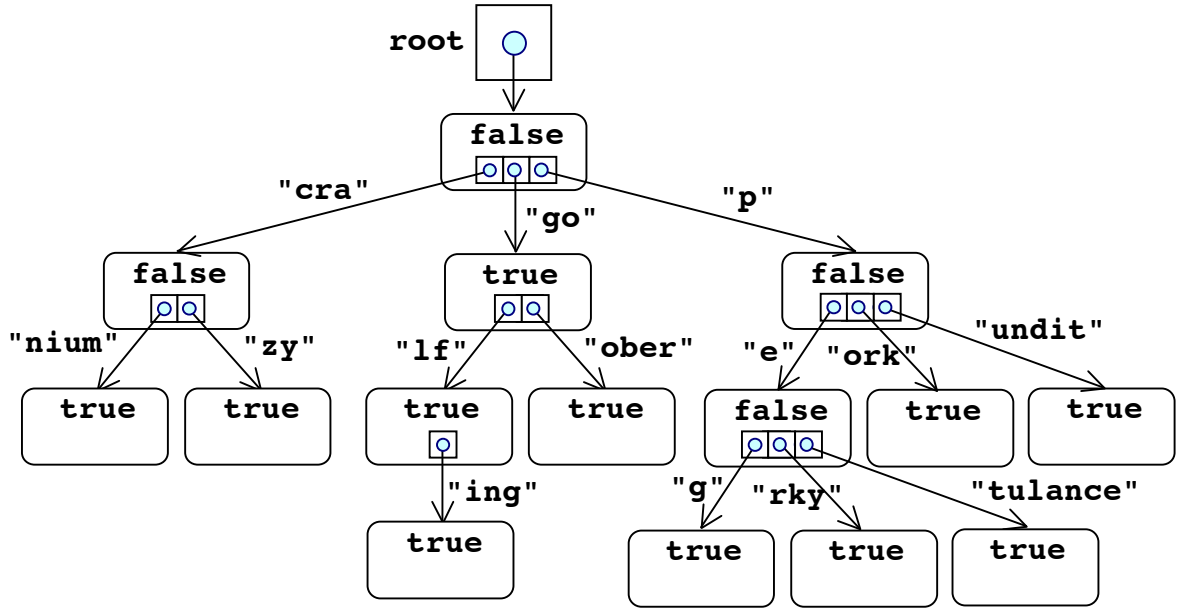
```
struct quadtree {
   int lowx, highx; // smallest and largest x value covered by node
   int lowy, highy; // smallest and largest y value covered by node
   bool isBlack; // entirely black? true.  Entirely white? False. Mixed? ignored
   quadtree *children[4]; // 0 is NW, 1 is NE, 2 is SE, 3 is SW
};
```

Assume the lower left corner of the image is the origin, and further assume the image is square and that the dimension is a perfect power of two.

```
static quadtree *gridToQuadtree(Grid<bool>& image);
```

Discussion Problem 2: Patricia Trees

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 on the preceding page 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 **containsWord** function, which accepts the root of a Patricia tree and a word, and returns **true** if and only if the supplied word is present. Even though the **connection**s descending from each node are sorted alphabetically, you should just do a **linear search** across them to see which one, if any, is relevant. Implement your function without recursion.

```
static bool containsWord(const node *root, const string& word);
```

Discussion Problem 3: Regular Expressions

A regular expression is a **string** used to match text. Regular expressions are, for our purposes, comprised of lowercase alphabetic letters along with the characters ***, +,** and **?**. In regular expressions, the lowercase letters match themselves. * is always preceded by an alphabetic character and matches zero or more instances of the preceding letter. **+** is similar to *, except that it matches 1 or more instances of the preceding letter. **?** states that the preceding letter may or may not appear exactly once.

Here are some regular expressions:

grape	matches **grape** as a word and nothing else
letters?	matches **letter** and **letters**, but nothing else
a?b?c?	matches **a**, **b**, **c**, **ab**, **ac**, **bc**, **abc**, and the empty string
lolz*	matches **lol**, **lolz**, **lolzz**, **lolzz**, and so forth
lolz+	matches **lolz**, **lolzz**, **lolzzz**, and so forth

All of the *****, **+** and **?** characters must be preceded by lowercase alphabetic letters, or else the regular expression is illegal.

Regular expressions play nicely with the trie data structure we began discussing in lecture on Monday. We'll use this exposed data structure to represent the trie:

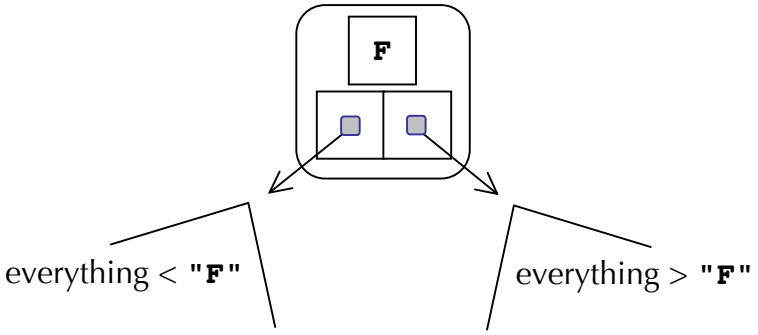
```
struct node {
   bool isWord;
   Map<char, node *> suffixes;
};
```

Write the **matchAllWords** function, which takes a trie of words (via its root node address) and a regular expression as described above, and populates the supplied **Set<string>**, assumed to be empty, with all those words in the trie that match the regular expression.

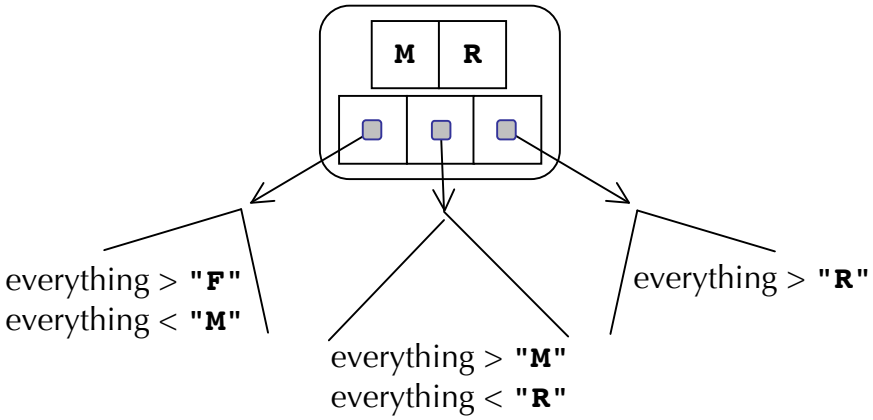
```
static void matchAllWords(const node *trie, const string& regex,
                          Set<string>& matches);
```

Lab Problem 1: Exponential Trees

Exponential trees are similar to binary search trees, except that the **depth** of the node in the tree dictates how many elements it can store. The root of the tree is at depth 1, so it contains 1 string and two children. The root of a tree storing **string**s might look like this:



If completely full, a node at depth 2—perhaps the right child of the root above—might look like this:



Generally speaking, a node at depth **d** can accommodate up to **d** strings. Those **d** strings are stored in sorted order within a **Vector<string>**, and they also serve to distribute all child elements across the **d + 1** sub-trees.

Exponential trees can be generalized to store any one type, but we'll stick to **string**s and avoid the template business. We will, however, commit you to the following data structure:

```
struct expnode {
   int depth;                     // depth of the node within the tree
   Vector<std::string> values;    // stores up to depth keys in sorted order
   expnode **children;            // set to NULL until node is saturated.
};
```

The lab project includes the definition of a **class** called **ExponentialTree**, which is a simple **string** set that's backed by the exponential tree structure we're describing here. The constructor, destructor, and **contains** method has already been implemented. Your job is to provide a working implementation of the **add** method. You'll want to update the **exponential-tree.cpp** file to include your code, and to use **exponential-tree-test.cpp** to exercise the full functionality. You may need to update **exponential-tree.h** if you elect to add some helper methods.

Some rules:

- Each node must keep track of its **depth**, because the depth alone decides how many elements it can hold, and how many sub-trees it can support.
- The **string** values are stored in the **values** vector, which maintains all of the **string**s it's storing in sorted order. We use a **Vector<string>** instead of an exposed array, because the number of elements stored can vary from **0** to **depth**.
- **children** is a dynamically allocated array of pointers to sub-trees. The **children** pointer is maintained to be **NULL** until the **values** vector is full, at which point the **children** pointer is set to be a dynamically allocated array of **depth + 1** pointers, all initially set to **NULL**. Any future insertions that pass through the node will actually result in an insertion into one of **depth + 1** sub-trees.