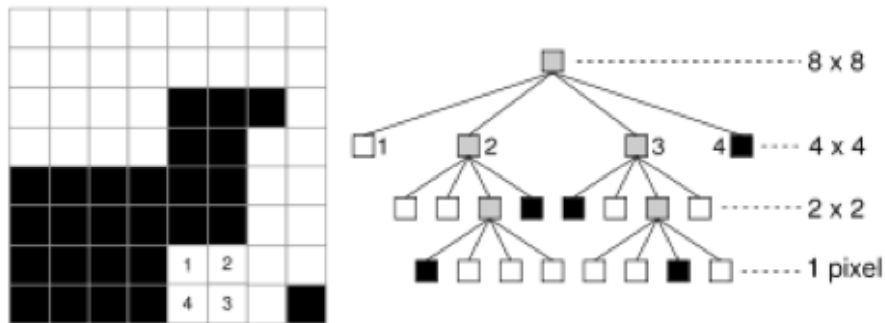


Section Handout

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:



The 8 pixel 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:

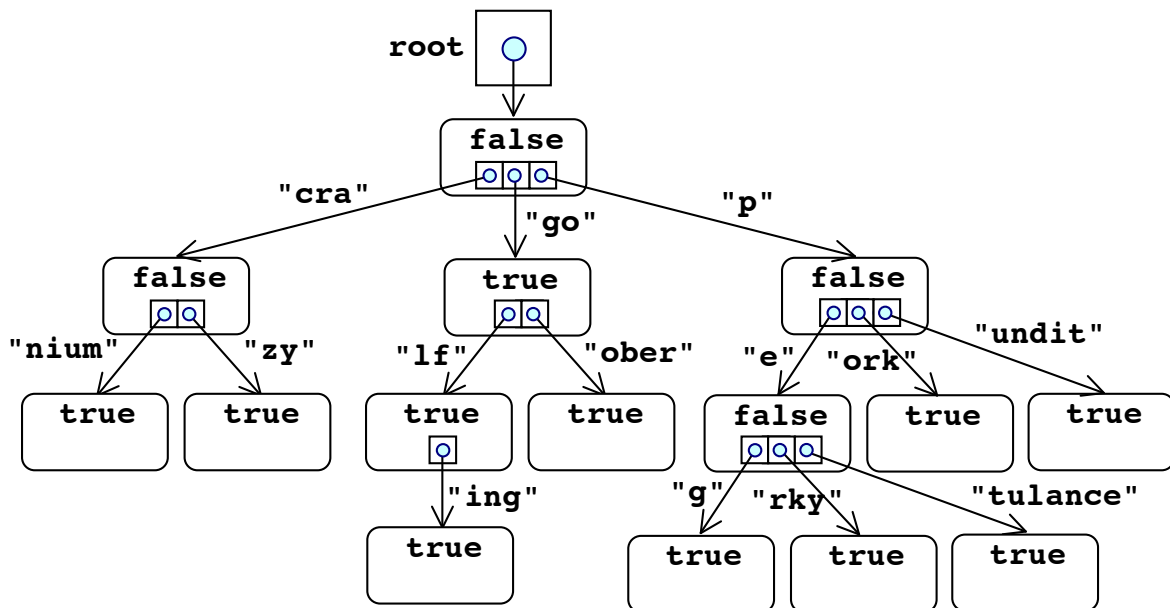
```
struct quadtree {
    int lowx, highx; // smallest and largest x value covered by node
    int lowy, highy; // smallest and largest y value covered by node
    bool isBlack; // entirely black? true. Entirely white? False. Mixed? ignored
    quadtree *children[4]; // 0 is NW, 1 is NE, 2 is SE, 3 is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

```
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 still at most 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.

The following 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
};
```

- First, 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 **connections** descending from each node are sorted alphabetically, you should still just do a **linear search** across them to see which one, if any, is relevant. Implement your function without recursion.

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

- Now write the **map** template function, which applies the supplied function to every single string—in alphabetical order—represented by the tree. The function is designed to take client data, represented by the **ClientData** template parameter. You'll probably want to write another wrapping function.

```
template <typename ClientData>
void map(node *root, void (mapfn)(string, ClientData&), ClientData& data);
```

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 , lolzzz , 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 Friday. We'll use this exposed data structure to represent the trie:

```
struct node {
    bool isWord;
    node *children[26];
};
```

Write the **findMatchingWords** 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.

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

Lab Problem 1: JavaScript Object Notation

JavaScript Object Notation, or JSON, is a popular, structured-data-exchange format that's easily read by humans and easily processed by computers. It's somewhat like XML in that both encode a hierarchy of information, but JSON is visually easier to take, and it's based on a small, fairly simple subset of JavaScript.

For the purposes of this week's lab, we're going to assume that the only primitive types of interest are integers, strings, and Booleans. We'll pretend we're in a world without fractions, and characters can just be represented as strings of length one. We'll further simplify everything so that:

- All integers are nonnegative, so you'll never encounter a negative sign. We'll assume that all integers are small enough that they can fit into a C++ **int** without overflowing memory.
- Strings are delimited by double quotes (real JSON also allows strings to be delimited by single quotes as well, but we'll pretend that's not the case here).
- The only Boolean constants are **true** and **false** (all lowercase, and no delimiting quotes).

Each of the following represents a legitimate JSON literal:

```
1 4124 4892014 true false "CS106X" "http://www.youtube.com" "hello there"
```

More interesting are the two composite types—the array and the dictionary.

The array is an ordered sequence much like our **Vector** is, except that it's heterogeneous and allows the elements to be of varying types. The full arrays are bookended by "[" and "] ", and array elements are separated by commas.

Here are some simple array literals, where all elements are of the same type:

```
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]
      [true, true, false, true]
      ["do", "re", "mi", "fa", "so", "la", "ti", "do"]
      [{"Rachel", "Finn"}, {"Brittany", "Artie"}, {"Puck", "Quinn"}]
```

But arrays can be truly heterogeneous, so that peer elements needn't be of the same type. That means the following also flies:

```
[1138, "star wars", false, [], [8, "C3PO", ["empire", [true], "R2D2"]]]
```

While the above array is contrived to illustrate heterogeneity, it's representative of the weakly typed programming languages that have blossomed over the past 20 years. (C++ is strongly typed, which is why everything needs to be declared ahead of time, and why any given C++ **Vector** can only store data of a single type.)

The JSON dictionary is little more than a serialization of a **Map**, save for the difference that the values needn't be all of the same type (heterogeneity once again). The dictionary literal is bookended by "{" and "}", and individual key-value pairs are delimited by commas. Each key-value pair is expressed as a key (with double quotes to emphasize the requirement that it be a string (in our version)), followed by a colon, followed by the JSON representation of the key's value. Here are some JSON dictionaries to chew on:

```
{"Bolivia": "Sucre", "Brazil": "Brasilia",
  "Colombia": "Bogota", "Argentina": "Buenos Aires"}

{"Hawaii": [808], "Arizona": [480, 520, 602, 623, 928], "Wyoming": [307],
  "New York": [212, 315, 347, 516, 518, 585, 607, 631, 646, 716, 718, 845, 914, 917],
  "Alaska": [907], "Louisiana": [225, 318, 337, 504, 985]}

{"Mike": {"Phone": "425-555-2912", "Hometown": "Midlothian, TX"},
  "Jerry": {"Phone": "415-555-1143", "Hometown": "Cinnaminson, NJ"},
  "Ben": {"Phone": "650-555-4388", "Hometown": "San Pedro, CA"}}
```

Any just in case it isn't clear, arrays can contain dictionaries as elements. None of the above examples included any, because I hadn't introduced dictionaries by then.

In the starter file bundle I've assembled for this week's lab, I've created a data files called **"json-lite-primitive-data.txt"** and **"json-lite-data.txt"**. The first file contains JSON that's correct parsed and printed by the starter code, and the second file is a superset of the first that includes some JSON that'll be recognized as legitimate once you finish lab.

Your job for lab this week is to write a collection of functions that parse JSON, build in-

memory versions of the JSON, print them out, and finally properly dispose of them. To get you started, I've defined the core data structure you'll be using, and I've provided implementations for the integers, Booleans, and strings. Your job is to:

- read up on **unions**, which are used by the implementation. I could spend a page talking about them, or you can get the skinny on them from your friendly section leader and your equally friendly Internet. Based on the starter code I've given you, you should be able to intuit how **unions** work without much trouble.
- understand why the in-memory versions of a JSON literal are actually trees when arrays and dictionaries are involved. It's that understanding that defends my decision to introduce JSON in a section handout on trees.
- complete implementations of the provided **parseJSON**, **prettyPrintJSON**, and **disposeJSON** functions. The implementations of each already handle the integer, Boolean, and text string primitives, and you're to extend them to handle arrays and dictionaries, paying careful attention to their heterogeneity. In the process you'll not only learn about JSON, memory, trees, pointers, and memory, but you'll also gain some insight into how compilers go about their business when reading in larger program files.

I've included a demo application that makes it clear from context what "pretty" printing is, but in general it means that everything is printed inline with a reasonable amount of intervening whitespace between array and dictionary entries, and between key-value pairs. If you'd like, you can spend some time trying to emulate how full blown pretty printing might work by trying to emulate the functionality produced at

<http://www.cerny-online.com/cerny.js/demos/json-pretty-printing>