

## Section Solution

---

### Discussion Problem 1 Quadtrees

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

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;
}
```

### 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.

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

    return curr->isWord;
}

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

```

        if (cmp < 0) break;
    }
    return -1;
}

```

- Provided you understand the recursive structure of the Patricia tree, the implementation of map is almost insultingly compact.

```

template <typename ClientData>
void map(node *root, void (mapfn)(string, ClientData&), ClientData& data) {
    if (root == NULL) return;
    map(root, mapfn, data, "");
}

template <typename ClientData>
void map(node *root, void (mapfn)(string, ClientData&),
        ClientData& data, string prefix) {
    if (root->isWord) mapfn(prefix, data); // by design, root is never NULL
    for (int i = 0; i < root->children.size(); i++) {
        map(root->children[i].subtree, mapfn, data,
            prefix + root->children[i].letters);
    }
}

```

### Discussion Problem 3 Solution: Regular Expressions

```

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

    if (root == NULL) return;
    if (regex.empty()) {
        if (root->isWord) {
            matches.add(workingPrefix);
        }
        return;
    }

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

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

```

## Lab Problem 1 Solution: JavaScript Object Notation

Here's the core of my own parsing solution. I decided parsing arrays and dictionaries was complicated enough that I created functions to manage the details.

```
// prototype necessary for mutual recursion
JSONElement *parseJSON(Scanner& s);
Vector<JSONElement *> *parseJSONArray(Scanner& s) {
    Vector<JSONElement *> *array = new Vector<JSONElement *>();
    bool firstElementConsumed = false;
    while (true) {
        string lookahead = s.nextToken();
        if (lookahead == "]" ) return array;
        if (firstElementConsumed && lookahead != ",") {
            Error("Oops! Commas need to separate elements in a JSON array.");
        } else if (!firstElementConsumed) {
            s.saveToken(lookahead);
        }

        JSONElement *element = parseJSON(s);
        firstElementConsumed = true;
        array->add(element);
    }
}

Map<JSONElement *> *parseJSONDictionary(Scanner& s) {
    Map<JSONElement *> *dictionary = new Map<JSONElement *>();
    bool firstEntryConsumed = false;
    while (true) {
        string lookahead = s.nextToken();
        if (lookahead == "}" ) return dictionary;
        if (firstEntryConsumed && lookahead != ",") {
            Error("Oops! Commas need to separate entries in a JSON dictionary.");
        } else if (!firstEntryConsumed) {
            s.saveToken(lookahead);
        }

        string key = s.nextToken();
        if (s.nextToken() != ":") {
            Error("Expected a colon to separate the key and value.");
        }

        JSONElement *value = parseJSON(s);
        firstEntryConsumed = true;
        dictionary->put(key, value);
    }
}

JSONElement *parseJSON(Scanner& s) {
    string lookahead = s.nextToken();
    if (lookahead.empty()) return NULL;

    JSONElement *element = new JSONElement;
    if (isdigit(lookahead[0])) {
        element->type = JSONElement::Integer;
        element->value.intValue = StringToInteger(lookahead);
    } else if (lookahead == "true" || lookahead == "false" ) {
        element->type = JSONElement::Boolean;
        element->value.boolValue = lookahead == "true";
    } else if (lookahead[0] == '"') {
```

```

        element->type = JMLElement::String;
        element->value.stringValue = new string(lookahead);
    } else if (lookahead == "[") {
        element->type = JMLElement::Array;
        element->value.arrayValue = parseJSONArray(s);
    } else if (lookahead == "{") {
        element->type = JMLElement::Dictionary;
        element->value.dictionaryValue = parseJSONDictionary(s);
    } else {
        Error("JSON element type passed to parseJSON not recognized.");
    }
}

return element;
}

```

Printing is also complex enough for arrays and dictionaries that I went with some helper functions as well.

```

void prettyPrintJSON(JMLElement *jsonRoot); // forward prototype
void prettyPrintJSONArray(Vector<JMLElement *> *array) {
    if (array->isEmpty()) {
        cout << "[]";
        return;
    }

    cout << "[";
    prettyPrintJSON(array->getAt(0));
    for (int i = 1; i < array->size(); i++) {
        cout << ",";
        prettyPrintJSON(array->getAt(i));
    }
    cout << "]";
}

void prettyPrintJSONDictionary(Map<JMLElement *> *entries) {
    if (entries->isEmpty()) {
        cout << "{}";
        return;
    }

    cout << "{";
    bool firstEntryPrinted = false;
    foreach (string key in (*entries)) {
        if (firstEntryPrinted) cout << ",";
        cout << key << ":";
        prettyPrintJSON(entries->get(key));
        firstEntryPrinted = true;
    }
    cout << "}";
}

void prettyPrintJSON(JMLElement *jsonRoot) {
    switch (jsonRoot->type) {
        case JMLElement::Integer:
            cout << jsonRoot->value.intValue;
            return;
        case JMLElement::Boolean:
            cout << (jsonRoot->value.boolValue ? "true" : "false");
            return;
        case JMLElement::String:

```

```

        cout << *(jsonRoot->value.stringValue);
        return;
    case JMLElement::Array:
        prettyPrintJSONArray(jsonRoot->value.arrayValue);
        return;
    case JMLElement::Dictionary:
        prettyPrintJSONDictionary(jsonRoot->value.dictionaryValue);
        return;
    }
}

```

Freeing memory is less complicated—uncomplicated enough that I didn't even bother writing helper functions.

```

void disposeJSON(JMLElement *jsonRoot) {
    switch (jsonRoot->type) {
        case JMLElement::Integer:
        case JMLElement::Boolean:
            // value elements don't themselves need to be freed,
            break;
        case JMLElement::String:
            delete jsonRoot->value.stringValue;
            break;
        case JMLElement::Array:
            for (int i = 0; i < jsonRoot->value.arrayValue->size(); i++) {
                disposeJSON(jsonRoot->value.arrayValue->getAt(i));
            }
            delete jsonRoot->value.arrayValue;
            break;
        case JMLElement::Dictionary:
            foreach (string key in (*jsonRoot->value.dictionaryValue)) {
                disposeJSON(jsonRoot->value.dictionaryValue->get(key));
            }
            break;
    }
    delete jsonRoot;
}

```