Eric Roberts
CS106B

Handout #54
March 11, 2013

Practice Final Examination #2

Review session: **Sunday, March 17, 3:00–5:00 P.M. (Hewlett 200)**

Scheduled finals: **Tuesday, March 19, 12:15–3:15 P.M. (Hewlett 200)**
Thursday, March 21, 12:15–3:15 P.M. (Hewlett 200)

1. Simple algorithmic tracing (5 points)

Diagram the binary search tree that results from adding—in order—the English words for the first ten counting numbers:

```
"one", "two", "three", "four", "five", "six", "seven", "eight", "nine", "ten"
```

to an empty binary search tree. As in the section problem, you should use the following definition for the nodes of the BST:

```
struct BSTNode {
   string key;
   BSTNode *left, *right;
};
```

Your diagram can be very simple and need show only the value of the key in each node and a line connecting that node to its left and right subtrees. In the problem, you can omit empty subtrees entirely.

2. Recursion (15 points)

Let's define an English word as ***reducible*** if it is possible to cross out one of its letters and still have an English word and, moreover, it is possible to repeat the process all the way to a single letter. As a simple example, the word *cats* is reducible because you can cross out first the *s,* then the *c,* and then the *t,* leaving the words *cat, at,* and *a,* in order. As a more extensive example, the longest reducible word in the `EnglishWords.dat` lexicon is *complecting* (the process of joining by weaving or binding together), which survives the following chain of deletions (some of which are admittedly unusual words):

complecting
completing
competing
compting
comping
coping
oping
ping
pig
pi
i

Write a function

```
bool isReducible(string word, Lexicon & english);
```

that takes an English word and a lexicon of English words and determines whether the word is reducible.

3. Heap-stack diagrams and memory tracing (10 points)

Suppose that you have a file containing the following code:

```
struct Cell {
   int value;
   Cell *link;
};

Cell *createIndexList(int n, Cell *tail = NULL);

int main() {
   int k = 2;
   Cell *list = createIndexList(k);
   return 0;
}

Cell *createIndexList(int n, Cell *tail) {
   Cell *cp = NULL;
   if (n == 0) {
      return tail;   ←Diagram memory at this point
   } else {
      cp = new Cell;
      cp->value = n;
      cp->link = tail;
      return createIndexList(n - 1, cp);
   }
}
```

Using the heap-stack diagrams from Chapter 12 as a model, draw a picture showing the contents of memory at the indicated point in the execution. You need not include explicit addresses in your diagram, but must indicate—either through addresses or arrows—where reference values point in memory. Your diagram should include the names of the stack variables.

heap | *stack*

4. Linear structures and hash tables (15 points)

Have you ever wondered how Google is able to search for sequences of words that you enter using quotation marks? For example, if you enter

in the search box, Google returns all the page references in which `"hash"` and `"table"` appear as consecutive words. Without the quotation marks, Google returns the set of pages on which both words appear, in any order, possibly in completely different parts of the page. Including the quotation marks makes it much more likely that Google's results include the information you're looking for.

It might at first seem as if Google must keep track of where every sequence of words appears, but that would require an impossibly large amount of storage, even for Google. What Google's web crawlers do instead is record the *position* of each word on the page along with URL of the page itself. For example, when Google looks up the pages on which the words `"hash"` and `"table"` appear, it gets lists that look something like the two columns in Figure 1 at the bottom of the page. The table shows, for example, that the word `"hash"` appears as the second word in **en.wikipedia.org/wiki/Hash_function**, the fifth and eighth words of **en.wikipedia.org/wiki/Hash_table**, and as word 2784 in the CS106B course reader at **cs106b.stanford.edu/Materials/Reader.pdf**. Similarly, the word `"table"` appears as word 23 on **www.w3.org/TR/CSS2/tables.html** and various other places, including the sixth word of **en.wikipedia.org/wiki/Hash_table** and word 2785 of the course reader.

Figure 1. Google entries for the words *hash* and *table*

Entries for `"hash"`

en.wikipedia.org/wiki/Hash_function	2
www.hashrestaurant.com/	3
en.wikipedia.org/wiki/Hash_table	5
en.wikipedia.org/wiki/Hash_table	8
en.wikipedia.org/wiki/Hashish	13
cs106b.stanford.edu/Materials/Reader.pdf	2784

Entries for `"table"`

www.w3.org/TR/CSS2/tables.html	23
en.wikipedia.org/wiki/Table	2
www.roundtablepizza.com	2
en.wikipedia.org/wiki/Hash_table	6
cs106b.stanford.edu/Materials/Reader.pdf	2785
www.amazon.com/Tables-Patio-Furniture/	17

The information in the two lists is all that Google needs to look for strings of consecutive words. When you look up **"hash table"** as a quoted string, all that Google has to do is find all the places in which **"hash"** appears and see which ones of those contain **"table"** in the next position. In the lists shown in Figure 1, there are two matches. The page **en.wikipedia.org/wiki/Hash_table** contains **"hash"** at position 5 and **"table"** at position 6. Similarly, **cs106b.stanford.edu/Materials/Reader.pdf** has **"hash"** at position 2784 and **"table"** at position 2785.

Suppose that you have access to the full index of Google's data structure in the form of a **Map< string,Vector<WebEntry> >**, where the type **WebEntry** is defined as follows:

```
struct WebEntry {
   string url;       /* The url in which the word appears   */
   int index;        /* The index of the word in that url   */
};
```

The keys in this map are English words, and the values are vectors of **WebEntry** structures showing where those words appear on the web. Thus, if you looked up **"hash"** in this map, you would get a vector of structures containing the values shown in the first column of Figure 1.

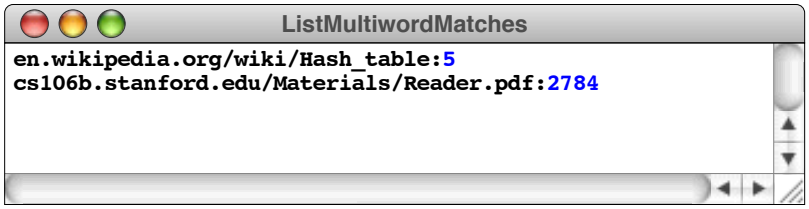
Your task in this problem is to write the function

```
void listMultiwordMatches(Vector<string> & words,
                          Map< string,Vector<WebEntry> > & webMap);
```

that takes two arguments: a vector of the search words (the words that were enclosed in quotation marks) and the map representing the web data base. The function should then display on the console a list of all the URLs on which those words appear consecutively on the page. Each line of the output should consist of the URL for the page, a colon, and the position number at which the match occurs. Thus, assuming that the information in Figure 1 is stored in a map called **webMap**, executing the code

```
Vector<string> words;
words.add("hash");
words.add("table");
listMultiwordMatches(words, webMap);
```

should generate the following output:



5. Trees (15 points)

Using the code for the expression evaluator (the same code you were given for the BASIC assignment) as a starting point, implement the method

```
string unparse(Expression *exp);
```

The result of **unparse** should be a string representation of the expression that includes parentheses only when the parser would need them to indicate the correct order of operations. For example, the expression

```
x + (2 * y)
```

should be unparsed as

```
x + 2 * y
```

because the parentheses are not necessary to indicate the desired order of operations, which is already determined by the fact that * has higher precedence than +. By contrast, the expression

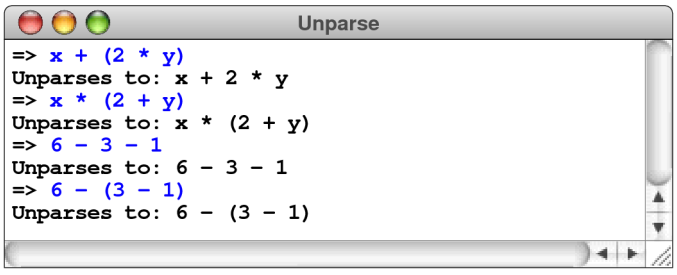
```
x * (2 + y)
```

must include the parentheses in the unparsed form.

The following main program illustrates the operation of **unparse**:

```
int main() {
   TokenScanner scanner;
   scanner.ignoreWhitespace();
   scanner.scanNumbers();
   while (true) {
      string line = getLine("=> ");
      if (line == "quit") break;
      scanner.setInput(line);
      Expression *exp = parseExp(scanner);
      cout << "Unparses to: " << unparse(exp) << endl;
      delete exp;
   }
   return 0;
}
```

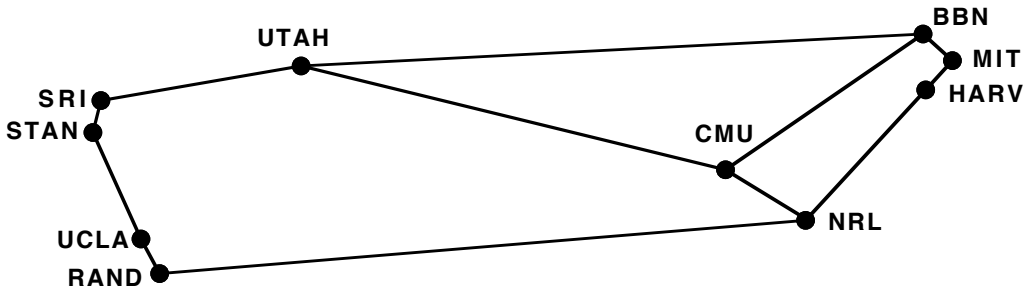
Running this program might produce the following sample run:



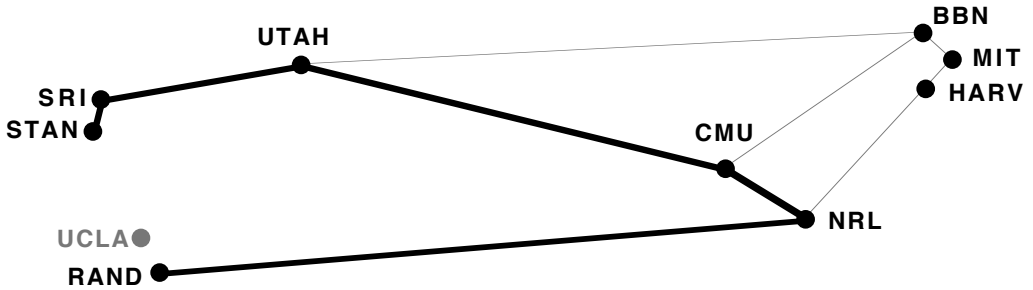
The last few lines of the sample run illustrate the fact that parentheses may be required even if two operators have the same precedence. As it is implemented in Chapter 19, the parser assumes that all operators are *left associative,* which means that the operator on the left takes precedence.

6. Graphs (15 points)

One of the important goals in the design of the ARPANET—the forerunner of today's Internet that you have seen in several sample problems—was to avoid having the failure of any node in the network break it into disconnected parts. Consider, for example, the early ARPANET diagram you have already seen on the first practice exam:



In this graph, none of the nodes represents a single point of failure. If, for example, the **`UCLA`** node were to go down, you could still route messages between the **`STAN`** and **`RAND`** nodes using the following more circuitous path:



A graph in which you can remove any node and still have a single connected graph is said to be ***biconnected***.

Using the definitions from the **`graph.h`** and **`graphtypes.h`** interfaces, write a function

```
bool isBiconnected(Graph<Node,Arc> & g)
```

that returns **`true`** if the graph **`g`** is biconnected, and **`false`** otherwise.

In writing your solution to this problem, you should keep the following points in mind:

- You may assume that the original graph **`g`** is connected and that it is undirected, in the sense that there are always arc in both directions between any pair of connected nodes. This assumption means that you don't have to check for a path in both directions.
- Note that the graph **`g`** is passed by reference to **`isBiconnected`** for efficiency. You are not allowed to change the graph that the client has supplied.
- You may find it useful to define a helper function similar to **`pathExists`** from Section #7 (Handout #43A). Feel free to use that code or to modify it as necessary.

7. Implementing a class (15 points)

Although we used it on quite a few occasions this quarter, we didn't get a chance to look at the implementation of the **Grid** class. One possibility for the internal representation of the **Grid** class is a dynamic array of dynamic arrays. Suppose, for example, that you have a **Grid<int>** containing the following 3×3 grid of integers:

8	1	6
3	5	7
4	9	2

The internal representation of the grid must keep track of the number of rows and columns, along with a pointer to a dynamic array holding each of the rows. That array, in turn, contains pointers to dynamic arrays of the base type, which in this case is the type **int**. Thus, if you were to declare this grid as a local variable on the stack and initialize it to the values shown, the internal state of memory would look like this:

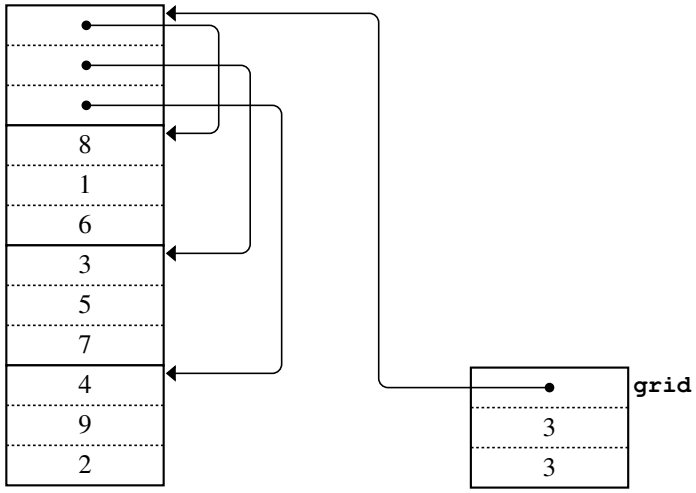


Figure 2 on the next page offers a stripped-down version of the **Grid** class that includes the constructor, the destructor, and the methods **get** and **set**, which get and set individual values. Note that the descriptions of **get** and **set** indicate that they call **error** if the row and column numbers are out of range. Your implementation must do so as well, using whatever error message you think is appropriate for that situation.

7a) Write the private section of the **grid.h** file. This section must includes any structure definitions and instance variables you need to represent the grid data structure using the array-of-arrays form illustrated in the diagram.

7b) Write the implementation section of the **grid.h** file. This section must include the code necessary to implement the constructor, destructor, **get**, and **set** methods as defined in the interface. Remember that **Grid** is a template class, which means that your method definitions must use the **template** keyword to define the base type. Feel free to write helper methods if it makes your job easier.

Figure 2. Partial listing of the `grid.h` interface

```
/*
 * File: grid.h
 * ------------
 * This interface exports a simplified version of the Grid class.
 */

#ifndef _grid_h
#define _grid_h

template <typename ValueType>
class Grid {

public:

/*
 * Constructor: Grid
 * Usage: Grid<type> grid(numRows, numCols);
 * -----------------------------------------
 * Defines a grid with the specified number of rows and cols.
 */

   Grid(int numRows, int numCols);

/*
 * Destructor: ~Grid
 * -----------------
 * Frees any heap storage associated with this grid.
 */

   ~Grid();

/*
 * Method: get
 * Usage: type value = grid.get(row, col);
 * ---------------------------------------
 * Returns the element at specified row/col location.  This method
 * raises an error if either index is outside of the grid boundaries.
 */

   ValueType get(int row, int col);

/*
 * Method: set
 * Usage: grid.set(row, col, value);
 * ---------------------------------
 * Replaces the element at the specified row/col location with a
 * new value, raising an error if either index is out of bounds.
 */

   void set(int row, int col, ValueType value);

private:

/* Add the private section here */

};

/* Add the implementation section here */

#endif
```