Eric Roberts
CS106B

Handout #55
June 1, 2009

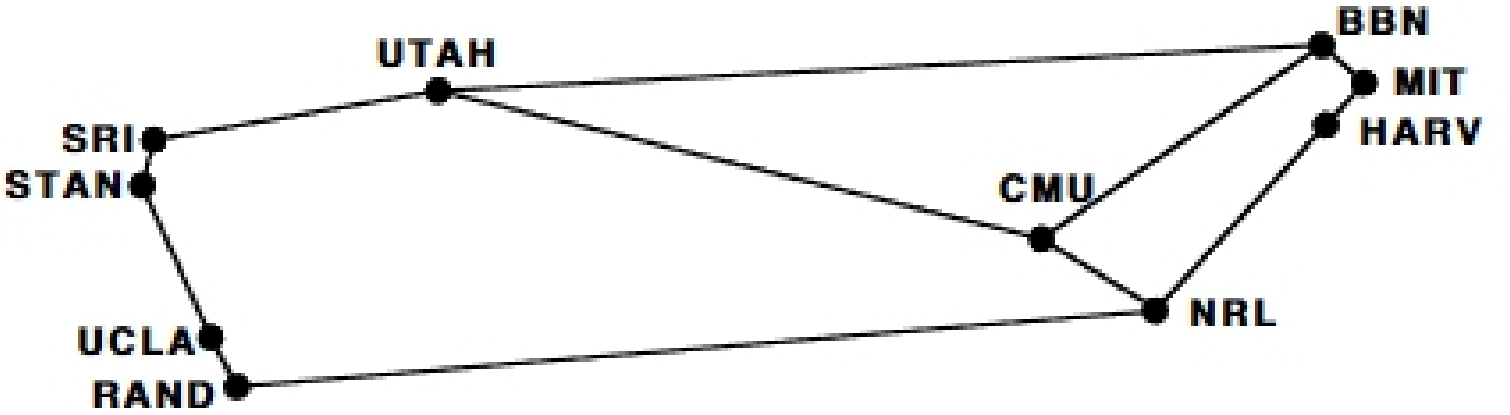
Practice Final Examination #2

Review session: **Wednesday, June 3, 7:00–9:00P.M. (Hewlett 201)**

Scheduled finals: **Friday, June 5, 8:30–11:30A.M. (Hewlett 200)**
Wednesday, June 10, 12:15–3:15P.M. (Hewlett 200)
Please remember that the final is open book

1. Simple algorithmic tracing (5 points)

The first ten nodes on the ARPANET (the forerunner to the modern Internet) were connected to form the following graph:

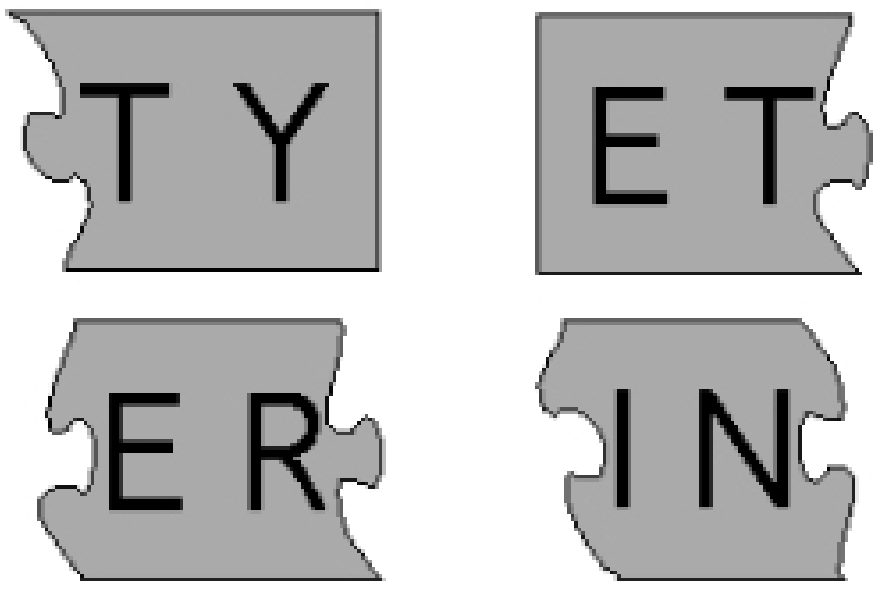


Assuming that node sets are processed in alphabetical order by name, in what order do the nodes get visited if you perform a depth-first search from the Stanford node (**STAN**)?

2. Recursion (15 points)

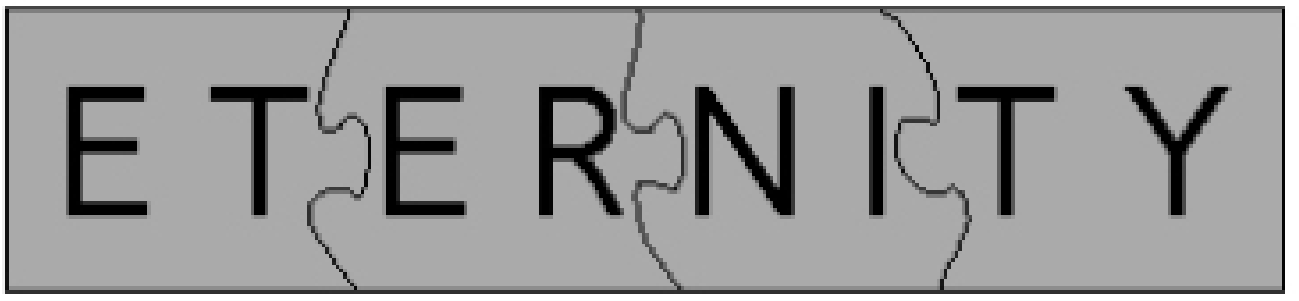
A decade ago this month, Christopher Monckton—a British puzzle enthusiast who also happens to be the heir to a hereditary peerage—released a puzzle called *Eternity* and offered a prize of one million pounds for a solution. The prize was claimed less than a year later by two mathematicians at Cambridge, Alex Selby and Oliver Riordan, who solved it with the aid of a computer. In July 2007, Monckton released *Eternity II* with a $2 million prize that remains unclaimed. See `http://uk.eternityii.com` for details.

While it is hardly possible to solve the actual Eternity puzzle in 15 minutes on a final exam, you can write a program to solve a simpler puzzle in a similar style. For this problem, your goal is to write a recursive solver for a one-dimensional jigsaw puzzle. You start with a set of pieces that might look like this:



Your goal is to see whether these pieces would fit inside a rectangular box, which is exactly the same height as the puzzle pieces. Because the **ET** and **TY** pieces each have a flat side, it is clear that these pieces must go at the ends of the box. With a little experimentation, you can also determine that the **ET** and **ER** pieces fit together nicely.

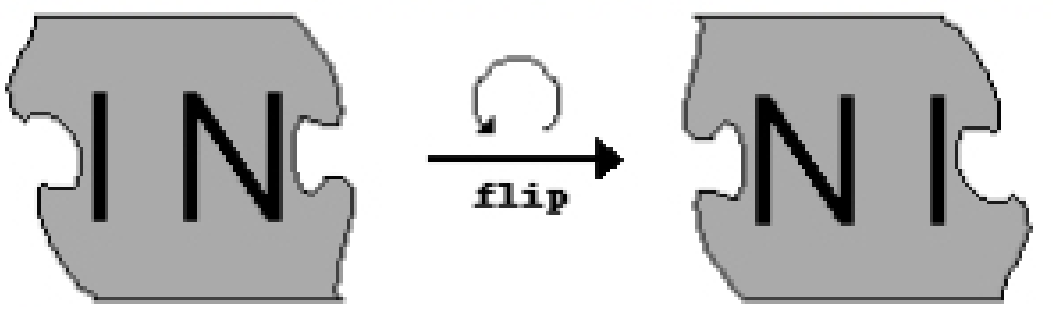
At first glance, however, it doesn't seem as if there is any place for the **IN** piece to go. But this a jigsaw puzzle, and it is perfectly fine to rotate the piece by a half turn, after which all the pieces fit together like this:



To turn this puzzle into a programming problem, suppose that you have been given a **puzzle.h** interface along with its corresponding implementation. The **puzzle.h** interface defines a class called **PuzzlePiece**, which exports the following methods:

```
bool attachesTo(PuzzlePiece & p);
bool hasFlatLeftSide();
PuzzlePiece flip();
```

The **attachesTo** method returns **true** if the piece on which it is called fits together with piece **p**, assuming that the pieces stay in their current orientation. For example, it the variables **pET** and **pER** contain **PuzzlePiece** objects representing the **ET** and **ER** pieces, calling **pET.attachesTo(pER)** would return **true**. The **hasFlatLeftSide** method, as its name suggests, returns **true** if that piece would fit against the left edge of the box, which means that **pET.hasFlatLeftSide()** would return **true**, but **pER.hasFlatLeftSide()** would return **false**. The **flip** method returns a new **PuzzlePiece** that looks like the original one except that it's been rotated 180 degrees. Thus, calling **flip** on a piece that looks like the **IN** piece in the original collection transforms it into a new piece, as follows:



Write a function

```
bool PuzzleIsSolvable(Set<PuzzlePiece> & puzzle);
```

that takes a set of values of type **PuzzlePiece** and returns **true** if those pieces fit into a one-dimensional rectangular box. If it is impossible to connect the pieces in a line, **PuzzleIsSolvable** should return **false**.

Remember that you don't have to define **PuzzlePiece** or any of its methods. Your job is to use these methods to implement **PuzzleIsSolvable**.

3. Linear structures and hash tables (15 points)

A hash table achieves its $O(1)$ average-case performance only if it remains fairly sparse. If you add enough keys to a table so that the bucket chains begin to get long, the performance of the hash table degrades. Chapter 12 discusses in general terms how one might solve this problem, but doesn't actually implement a solution.

Add a method

```
void rehash(int nBuckets);
```

to the **Map** class that rehashes all the keys from the map into a new bucket array whose size is given by the **nBuckets** parameter.

In writing your implementation, you should keep the following points in mind:

- You are adding this method to the implementation of the class and therefore have access to the private instance variables.
- The implementation of a method for a template class has a more complex prototype, but your knowledge of that syntax is not what we're testing here. In the **mapimpl.cpp** file, the header for **rehash** will look like this:

```
template <typename ValueType>
void Map<ValueType>::rehash(int nBuckets)
```

- You can't simply copy the contents of the old array into the new one, because the keys in the map will presumably hash to different buckets. You instead need to go through each of the key/value pairs and insert them into the new bucket array.

4. Function pointers (10 points)

Given the iterator facility, it is easy to write a method that returns the longest key (or any of the longest keys, if there are several that are of the same length). All you have to do is run the iterator through the keys and keep track of the longest one.

In this problem, you task is to implement

```
string LongestKey(Map<string> map);
```

that does not use iterators but instead uses the **mapAll** method to find the longest key. In this problem, remember that you are working as a client of the **Map** class and have no access to the internal structure.

5. Trees (15 points)

Write a function

```
bool ExpMatch(expressionT e1, expressionT e2);
```

that returns **true** if **e1** and **e2** are matching expressions, which means that they have exactly the same structure, the same operators, the same constants, and the same identifier names in the same order. If there are any differences at any level of the expression tree, your function should return **false**.