Eric Roberts
CS106B

Handout #52
March 8, 2013

Practice Final Examination

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

Scheduled finals: **Tuesday, March 19, 12:15–3:15P.M. (Hewlett 200)**
Thursday, March 21, 12:15–3:15P.M. (Hewlett 200)
Please remember that the final is open book

This handout is intended to give you practice solving problems that are comparable in format and difficulty to those that will appear on the final examination. A solution set to this practice examination will be handed out on Monday, along with a second practice exam. The solutions to the second practice exam will go out next Wednesday.

Time of the exam

The final exam is scheduled at two different times during exam period, as shown at the top of this handout. You may take the exam at either of the two scheduled times and need not give advance notice of which exam you plan to take. If you are unable to take the exam at either of the scheduled times or if you need special accommodations for the exam, please send an e-mail message to `eroberts@cs` stating the following:

- The reason you cannot take the exam at the scheduled time or the accommodations that you require.
- A time period during exam week at which you could take the exam. This time must be during the regular working day and must therefore start between 8:30 and 2:00.

In order to take an alternate exam, I must receive this message from you by 5:00P.M. on Wednesday, March 13. Replies will be sent by electronic mail on Friday, March 15.

Review session

There will be a review session on Sunday, March 18, from 3:00 to 5:00P.M. At the review session, I will announce the winners of the BASIC Contest and hold a random drawing for a special-prize winner chosen among those of you who have entered contests along the way.

Coverage

The exam covers the material presented in class through next Monday, which means that you are responsible for all the chapters in the reader except Chapter 20. Although you will certainly need to know the material from the earlier chapters—and there will be a recursion problem along the lines of Assignment #3—the exam concentrates on the material from the lectures after the midterm, most of which focus on implementing the collection classes.

General instructions

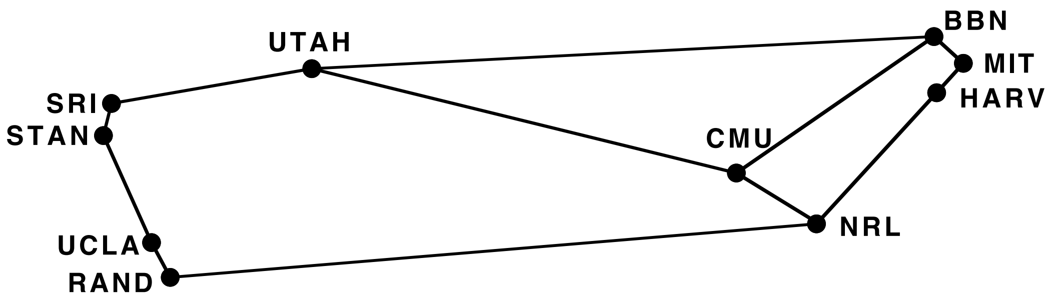
The instructions that will be used for the actual final look like this:

> Answer each of the questions given below. Write all of your answers directly on the examination paper, including any work that you wish to be considered for partial credit.
>
> Each question is marked with the number of points assigned to that problem. The total number of points is 90. We intend that the number of points be roughly equivalent to the number of minutes someone who is completely on top of the material would spend on that problem. Even so, we realize that some of you will still feel time pressure. If you find yourself spending a lot more time on a question than its point value suggests, you might move on to another question to make sure that you don't run out of time before you've had a chance to work on all of them.
>
> In all questions, you may include any functions or class definitions that are used on the handouts or in the reader and need not specify the `#include` line, if any, in which those definitions appear.
>
> The examination is open-book, and you may make use of any texts, handouts, or course notes. You may not, however, use a computer of any kind.

Note: To conserve trees, I have eliminated space for the answers from the practice exams. The actual final will have room for your answers and for any scratch work.

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, in what order do the nodes get visited if you perform a recursive depth-first search from the Stanford node (`STAN`)?

2. Recursion (15 points)

Most operating systems and many applications that allow users to work with files support ***wildcard patterns,*** in which special characters are used to create filename patterns that can match many different files. The most common special characters used in wildcard matching are `?`, which matches any single character, and `*`, which matches any sequence of characters. Other characters in a filename pattern must match the same character in a filename. For example, the pattern `*.*` matches any filename that contains a period, such as `US.txt` or `pathfinder.cpp`, but would not match filenames that do not contain a period. Similarly, the pattern `test.?` matches any filename that consists of the name `test`, a period, and a single character; thus, `test.?` matches `test.h` or `test.c`, but not `test.cpp`. These patterns can be combined in any way you like. For example, the pattern `??*` matches any filename containing at least two characters.

Write a function

```
bool filenameMatches(string filename, string pattern);
```

that takes two strings, representing a filename and a wildcard pattern, and returns `true` if that filename matches the pattern. Thus,

```
filenameMatches("US.txt", "*.*")        returns true
filenameMatches("test", "*.*")          returns false
filenameMatches("test.h", "test.?")     returns true
filenameMatches("test.cpp", "test.?")   returns false
filenameMatches("x", "??*")             returns false
filenameMatches("yy", "??*")            returns true
filenameMatches("zzz", "??*")           returns true
```

This problem is easier than it might seem at first, if you choose the right recursive decomposition. To help lead you in that direction, start by thinking about the first character in the pattern and distinguish the following cases:

- The pattern string is empty and therefore has no first character.
- The first character in the pattern string is a `?`.
- The first character in the pattern string is a `*`.
- The first character in the pattern string is any other character.

In each of these cases, think about what the `filename` string would have to look like in order to be a match. As in any recursive formulation, the strategy you choose has to reduce complex problems into simpler instances of the same problem.

In writing your answer to this problem, you should keep the following ideas in mind:

- The period (`.`) that typically separates parts of a filename is not a special character as far as `filenameMatches` is concerned. A period in `pattern` means that there must be a period in `filename`, which is exactly what happens with any other character.
- The `*` character can match any substring in `filename`, including an empty substring of length 0.
- If you find yourself getting bogged down in lots of detailed code, you are probably veering off onto the wrong track. Go back and look at the hints on recursive decomposition presented earlier in this problem and think again about your strategy.

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

The code in Figure 1 on the next page shows part of the implementation of a numeric binary search tree in which the keys are of type **`double`** rather than **`string`**. Using the heap-stack diagrams from Chapter 12 as a model, draw a picture showing the contents of memory just before the function **`insertKey`** returns from the point of its most deeply nested call (the recursive call generated by the second call to **`insertKey`** from the main program). 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*

Figure 1. Heap-stack tracing exercise

```
/*
 * File: NumericBST.cpp
 * --------------------
 * This file contains code for creating a numeric BST.
 */

#include <iostream>
using namespace std;

/*
 * Type: NumericBSTNode
 * --------------------
 * This type represents a node in a BST containing doubles.
 */

struct NumericBSTNode {
   double key;
   NumericBSTNode *left, *right;
};

/* Function prototypes */

void insertKey(NumericBSTNode * & t, double key);

/* Main program */

int main() {
   NumericBSTNode *numTree = NULL;
   insertKey(numTree, 3.14159);
   insertKey(numTree, 2.71828);
   return 0;
}

/*
 * Function: insertKey
 * Usage: insertKey(t, key);
 * -------------------------
 * Inserts the specified key at the correct location in the
 * binary search tree rooted at t.
 */

void insertKey(NumericBSTNode * & t, double key) {
   if (t == NULL) {
      t = new NumericBSTNode;
      t->key = key;
      t->left = t->right = NULL;
   } else {
      if (key != t->key) {
         if (key < t->key) {
            insertKey(t->left, key);
         } else {
            insertKey(t->right, key);
         }
      }
   }
}
```

4. Linear structures and hash tables (15 points)

A hash table can achieve its ideal *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 15 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 **`HashMap`** 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.
- A method for a template class has a relatively complex prototype, but your knowledge of that syntax is not what we're testing here. In the **`hashmapimpl.cpp`** file, the header for **`rehash`** will look like this:

  ```
  template <typename KeyType,typename ValueType>
  void Map<KeyType,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 hash to different buckets. You instead need to go through each of the key/value pairs and insert them into the new bucket array.

5. Trees (15 points)

When implementing a binary search tree, an important efficiency concern is taking care that the tree remains balanced to ensure logarithmic performance for insert and lookup. Although the course reader describes how to implement self-adjusting trees such as AVL, I did not cover the details in class. As it turns out, strategies that continually make minor rearrangements to rebalance the tree often end up spending too much time on those operations. An alternative strategy is to let the client determine whether a problem exists and, if so, rebalance the tree all at once.

One reasonably efficient strategy for rebalancing a tree is to transfer the nodes into a sorted vector and then reconstruct an optimally balanced tree from the vector. If you adopt this approach, you can implement a rebalancing operation for the **`BSTNode`** type by decomposing the problem into two helper methods as follows:

```
void rebalance(BSTNode * & root) {
   Vector<BSTNode *> v;
   fillVector(root, v);
   root = rebuildTree(v, 0, v.size() - 1);
}
```

The helper method

```
void fillVector(BSTNode *node, Vector<BSTNode *> & v);
```

adds all the nodes in the subtree rooted at `node` to the vector `v`, making sure that the nodes are added in the order they appear in the binary search tree. The helper method

```
BSTNode *rebuildTree(Vector<BSTNode *> & v, int start, int end);
```

recreates a tree by adding all the nodes from the vector `v` between the indices `start` and `end`, inclusive. Here, the goal is to make sure that the subtree returned by this operation is as balanced as possible, which means that the root must be as close as possible to the center of the range.

Write code for `fillVector` and `rebuildTree` to complete the implementation of the `rebalance` method.

In writing this problem, you should keep the following ideas in mind:

- Your implementation should not create or destroy any existing nodes, but simply rearrange them into a more balanced structure.
- For full credit, your implementation must run in $O(N)$ time, where N is the number of nodes. As always, this calculation does not take into account infrequent operations (such as the various versions of `expandCapacity`) whose amortized cost is constant. Thus, you may assume that adding an element to the end of a vector is a constant-time operation; adding an element at the beginning of a vector is not.

Woman Discovers Her Husband's Other Wife on Facebook
A Washington woman got quite a surprise when Facebook's "People You May Know" feature reportedly recommended a potential contact. The two women certainly had something in common: a husband.

—*Time,* March 13, 2012

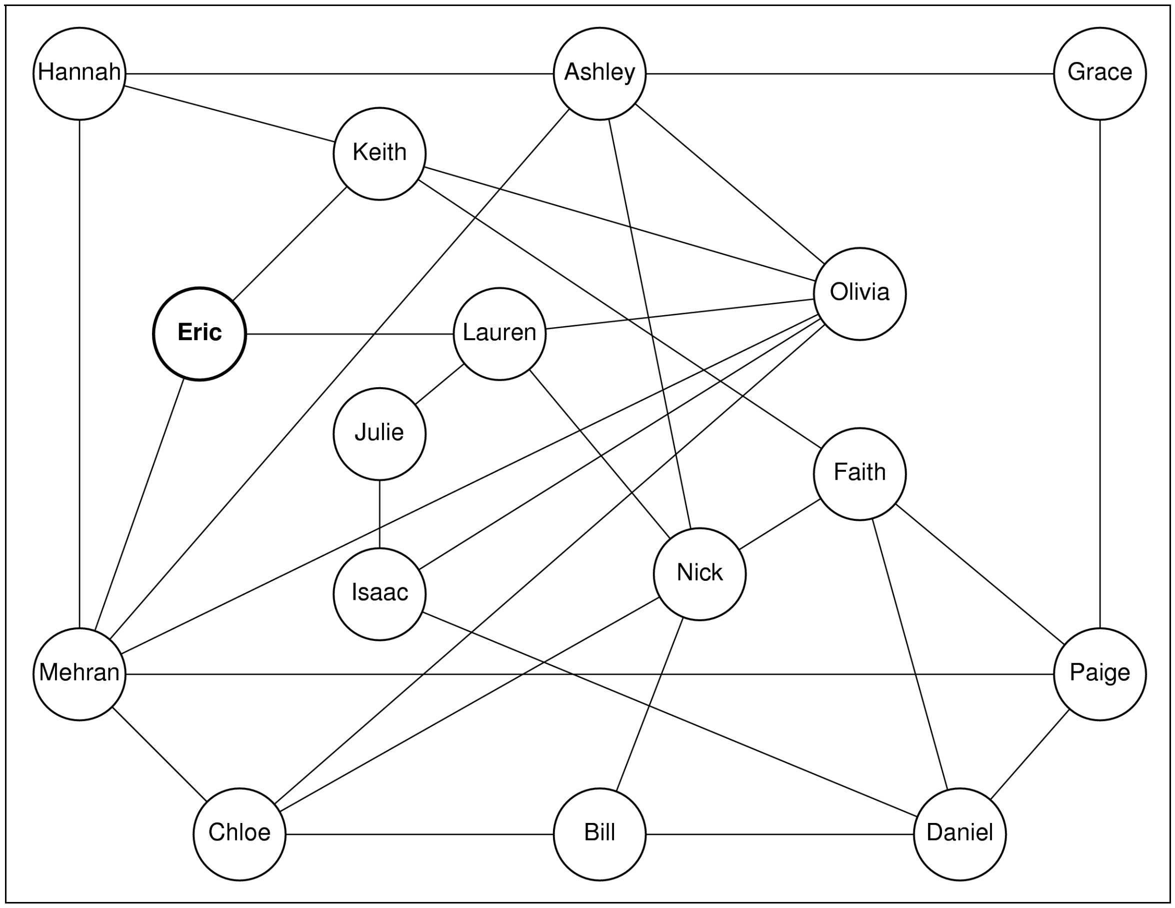
6. Graphs (15 points)

Although the feature is not nearly as dramatic as this past week's news story might suggest, Facebook does offer to find new friends by scanning its database to find people to whom you are connected by some number of mutual friends. To illustrate this process, imagine that one tiny part of the Facebook friend database consists of the graph shown in Figure 2, where friends are connected by an arc. For example, the arcs leading out of the boldface **Eric** node near the left side of the graph show that I am friends with Keith, Lauren, and Mehran. Given this graph, Facebook would probably suggest that I become friends with Olivia because she is directly connected to all three of my current friends. Facebook might also suggest that I might want to become friends with Hannah, because Hannah and I have two mutual friends: Keith and Mehran.

Write a function

```
void suggestFriends(Graph<Node,Arc> & g, Node *person);
```

Figure 2. Example of a friend network

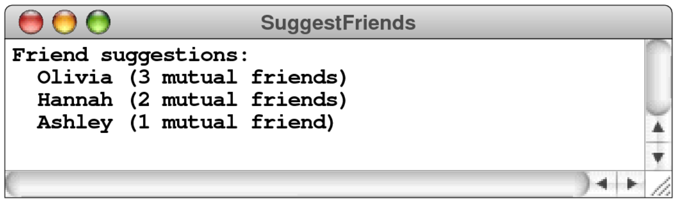


that writes out three (or fewer if there aren't three possibilities) suggested friends for the person at the specified node. The suggestions should be sorted in descending order by the number of friends the person has in common with each suggested friend.

For example, if `friendGraph` is initialized to the graph in Figure 3, calling

```
    suggestFriends(friendGraph, friendGraph.getNode("Eric"));
```

should generate output that looks like this:



```
Friend suggestions:
  Olivia (3 mutual friends)
  Hannah (2 mutual friends)
  Ashley (1 mutual friend)
```

There are other possibilities besides Ashley for the final name on this list. If several potential friends have the same count of mutual friends, your program is free to choose any order.

It might seem that one way to implement this problem would be to iterate over all nodes in the graph and count the number of mutual connections. The problem with this approach is that the Facebook friendship graph is *huge.* In this problem, you must therefore work only in the neighborhood of the initial node. You can certainly look at friends of friends, but not at completely unrelated parts of the graph.

As you write this program, you may find the following suggestions helpful:

- The **`Graph`** class and the **`Node`** and **`Arc`** types are the ones you were given for the Pathfinder assignment. You are not allowed to change any of these definitions, and must therefore use them as a client.
- The code will be easier to write if you include a helper function that counts the number of mutual friends given two nodes in a graph.
- Using the **`PriorityQueue`** class eliminates the need to sort the final list.

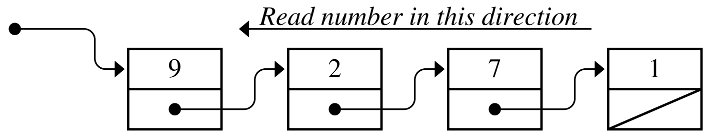
7. Implementing a class (15 points)

On most machines, the data type **`long`** is stored using 64 bits, which means that the largest positive value of type **`long`** is $2^{63}–1$, or 9,223,372,036,854,775,807. While this number seems enormous, there are some applications that require even larger integers. For example, if you were asked to compute the number of possible arrangements for a deck of 52 cards, you would need to calculate 52!, which works out to be

80,658,175,170,943,878,571,660,636,856,403,766,975,289,505,440,883,277,824,000,000,000,000

If you are solving problems involving integer values on this scale (which come up often in cryptography, for example), you need to use a software package that provides ***extended-precision arithmetic,*** in which large integers are represented in a form that allows them to grow dynamically.

Although there are more efficient techniques for doing so, one strategy for implementing extended-precision arithmetic is to store the individual digits in a linked list. In such representations, it is conventional—mostly because it makes implementing arithmetic operators easier—to arrange the linked list so that the units digit comes first, followed by the tens digit, then the hundreds digit, and so on. Thus, to represent the number 1729 as a linked list, you would arrange the linked-list cells in the following order:



In C++, it is almost always appropriate to encapsulate this sort of pointer-based structure inside a class definition so that the class can take care of memory management issues. Here, the best thing to do would be to define a class called **`BigInt`** that uses this linked list representation internally to represent an arbitrarily large interface. A part of that interface appears in Figure 3 on the next page. A complete interface would, of course, include other methods including the arithmetic operators, but the constructor, destructor, and **`toString`** methods shown in the figure offer at least a starting point.

Figure 3. Partial listing of an interface to support extended-precision integers

```
/*
 * File: bigint.h
 * --------------
 * This interface exports the BigInt class, which makes it
 * possible to represent integers of arbitrary magnitude.
 */

#ifndef _bigint_h
#define _bigint_h

#include <string>

class BigInt {

public:

/*
 * Constructor: BigInt
 * Usage: BigInt bigint(str);
 * --------------------------
 * Creates a new BigInt from the digits in the specified string,
 * which may begin with a minus sign to indicate a negative value.
 */

   BigInt(std::string str);

/*
 * Destructor: ~BigInt
 * Usage: (usually implicit)
 * -------------------------
 * Frees the memory used by a BigInt when it goes out of scope.
 */

   ~BigInt();

/*
 * Method: toString
 * Usage: string str = bigint.toString();
 * --------------------------------------
 * Converts a BigInt object to the corresponding string.
 */

   std::string toString();

. . . more entries not described here . . .

private:

. . . the contents of the private section go here . . .

};

#endif
```

7a) Write the private section for the **bigint.h**, which must include whatever structure definitions and instance variables you need to represent an integer in this linked-list-of-digits form.

7b) Write the **bigint.cpp** file necessary to implement the constructor, destructor, and **toString** methods as defined in the interface.

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

- Your internal representation for **BigInt** must use the linked-list format specified in the problem description.
- As implied by the comments for the constructor, the value stored in a **BigInt** can be either positive or negative. As a result, your internal data structure will have to include an instance variable to store the sign of the value along with the linked list of digits. The fact that the value is signed will also affect your implementation of **toString** to ensure that negative values are preceded by a negative sign.
- You should give at least some thought as to how you will represent the **BigInt** corresponding to the integer 0. In particular, you should make sure that calling **toString** on a zero value returns the string **"0"** rather than the empty string.