CSCI 333 Fall 2002
Project 2

This project will be completed in several steps.
Do not get behind!

Stage 1: Due October 25, 2002 – 12:15 pm

Implement a data structure to represent a Unix process "family". All members of the family are known by positive numbers. The family has a single *parent* and a, possibly empty, collection of *children*.

Here is a class definition for the *public* part of the data structure.

```
class IntFamily {
public:
      IntFamily(int Parent) ;       // Creates family with no children
      void addChild(int NewChild) ; // Adds new child to the family
      int getParent() const ;       // Returns parent of family
      int getSize() const ;         // Returns size of family
      int getChild(int I) const;    // Returns I'th child of family
      ~IntFamily() ;                // deconstructor
}
```

The first stage of the project is to be done *individually* by each student. Although you can implement this class totally from "scratch," your task should be easier if you use the list classes of the textbook.

Once you have written this code you should store it in the two files `IntFamily.h` and `IntFamily.cpp` in a subdirectory `csci/333/P2Stage1` of your UNCA CSCI Unix account.

Stage 2: Due 8 November, 2002 – 12:15 pm

Now write a C++ routine called `ParseFamily` that receives a single parameter, an opened `istream`. Read from that `istream` lines of the following format:

```
1 -> 11, 13, 15, 17, 19
2 -> 12, 14, 16, 18, 20
333 ->
```

Each of these lines will represents an `IntFamily` object. Your program should read each input line, transform it into an `IntFamily` object, and then insert that `IntFamily` into a list that implements textbook's `List` ADT. When an end-of-file condition is encountered on the istream, your program should return the list it has generated.

Most of `ParseFamily` has already been written for you. The CSCI 333 Projects page contains links to all the code you'll need to get started on this stage of the project.

You are allowed to choose one of two implementations of `ParseFamily`. One implementation uses the textbook's list class. There the prototype for `ParseFamily` is:

```
List<IntFamily> *ParseFamily(istream *inputStream) ;
```

The other implementation uses the vector class from the C++ STL (Standard Template Library). Here's its prototype:

```
vector<IntFamily> *ParseFamily(istream *inputStream) ;
```

Whichever way you choose, you'll need to add a few lines of code to do some error checking and child processing. Start working on this project today, or better yet, yesterday, because Stage 3 is much harder and you bet start working on it before Stage 2 is due.

You may do this stage of the project in groups of up to three students. Store your code in a file called `ParseFamily.cpp` in a subdirectory `csci/333/P2Stage2` of your UNCA CSCI Unix account.

Stage 3: Due 18 November, 2002 – 12:15 pm

Page 194 of the textbook gives an ADT for a general tree node. For stage 3, you are going to make two significant modifications to that program. All the code required to start stage 3 is available from the CSCI 333 Projects home page. The main routine, stored in the file `Proj2Stage3.cpp` will read the contents of a file, as specified in Stage 2, and output lines showing each node of the tree along with its descendents. For example, if the descendents of node 15 are 7 and then 3 and then 1, the program will write the line:

```
15 <- 7 <- 3 <-1
```

You may want to compile and run this program right now. It will make reading the rest of this description a bit easier.

The program runs correctly if the input conforms to the following restrictions:

1) All the integers of the families must be greater than 1.
2) The first parent must be the number 1.
3) No integer, excepting 1, can appear as a parent before it appears as a child.
4) Every integer is the parent of only one family.
5) No integer is, via some chain of offspring, its own parent.

Unfortunately, the program does not check for these conditions and can even "crash" if they are violated. Your task in Stage 3 is to add the code to check for these conditions.

You may do this stage of the project in groups of up to three students. Store your code in a file called `Proj2Stage3.cpp` in a subdirectory `csci/333/P2Stage3` of your UNCA CSCI Unix account.

Stage 4: Due 25 November, 2002 – 12:15 pm

The second problem with the program is that stores pointers to all the nodes in the family in an array `IntPeople` of 1000000 elements. If its input file contains an integer larger than 999999, the program will probably crash. Also, this represents a great deal of wasted space. In Stage 4 you are to use a better data structure, either a hash table as described in a book or the C++ `map` class to implement the association of integers to corresponding `GTNode`'s.

My strong suggestion is that you consider the C++ `map` class. Indeed it is *not* described in the textbook, but *in the real world* software developers have to learn, study, and use new modules all the time.

Again you may do this stage of the project in groups of up to three students. Store your code in a file called `Proj2Stage4.cpp` in a subdirectory `csci/333/P2Stage4` of your UNCA CSCI Unix account.

Stage 5: Due 6 December, 2002 – 12:15 pm

With your Stage 4 program, you are able to construct a graph from an input file describing the tree. In Stage 5, we are going to add a second file for making queries about the parentage of nodes in the tree.

The name of the second file will be given as the second argument to your program, now called `Proj2`. Each input line of this file will contain two comma-separated numbers. Here's an example input file:

```
13, 14
3001, 2001
```

For each input pair, your program should find the nearest common ancestor and the "distance" to that ancestor for either pair. For example, if the first input file contained the lines:

```
1 -> 15
15 -> 35, 45
45 -> 60
```

the nearest common ancestor of the pair 35 and 60 would be 15. The distance for 35 would be one. The distance for 60 would be two. In this case, your program should output the following line to report this information:

```
35 <-* 15 in 1, 60 <-* 15 in 2
```

Sometimes, one of the pair will be the direct ancestor of the other. In that case, there is a distance of zero to one of the ancestors. For example, if your program was asked for the nearest common ancestor of 15 and 60, it should reply

```
15 <-*15 in 0, 60 <-* 15 in 2
```

Here's a suggestion for the solving the program. Given a pair of numbers, first determine the distance of each to node 1, the root. For example, 35 is distance two to the root while 60 is distance three. Second, move down the tree from the most distant node, until you are the same level from the root. In the case, you'd move down one node, from 60 to 45. Now you have two pointers to nodes within the tree. Finally, move those two pointers down the tree until they encounter their nearest common ancestor.