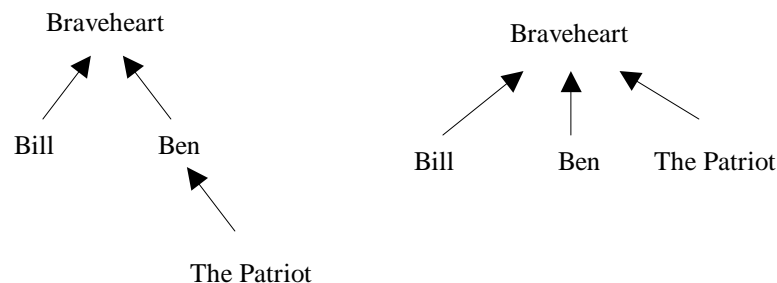


## Description

If you have ever visited an e-commerce website such as Amazon.com, you have probably seen a message of the form “people who bought this book, also bought these books” along with a list of books that other people have bought. This is an example of a recommender system. In this case, the purpose of the recommender is to bring to your attention to books that other people have bought, under the assumption that if you want to buy the book you are looking at you might also want to buy books that the other people bought. Of course, this assumption is often not a good one (we might have two people who would both buy books on C++ programming, but one likes historical novels about Scotland and the other likes to read nonfiction books about politics). So, Amazon.com is using a naïve approach to recommendation.

In this assignment, we will also use a naïve assumption for recommending movies. In our case, we will consider people and movies, and store information about whether a particular person likes a movie or not. Our recommendation system will take a person and a movie as a query, and answer “yes” or “no” if we would predict that the person would like the movie (of course, if the pair was already given to us as data, then it should answer “yes”). The assumption that we will use is that the “likes” relation is transitive in the following way: if Bill likes “Braveheart”, Ben likes “Braveheart”, and Ben likes “The Patriot”; then we would predict that Bill likes “The Patriot”. Of course, this assumption is as naïve as the one used by Amazon.com, but we can implement it nicely using the UNION/FIND algorithm on general trees.

In class, we talked about using the parent pointer implementation of general trees to test equivalence of objects by an equivalence relation, which is given by pairs of objects. Here we will abuse this idea, and use the data structure to solve the recommender problem. The abuse is that the “likes” relation as an irreflexive, asymmetric and transitive relation between objects from two different sets, and we will treat it as if it were reflexive and symmetric over one set. In particular, we will define the tree so that if a person A likes a movie B, then A and B occur in the same tree. For the



example above, the pairs would be (Ben,Braveheart), (Bill,Braveheart), (Ben, The Patriot), and two of the possible trees for these pairs are as follows. Using either of these trees, we can answer the query “does Bill like ‘The Patriot’?” by using the find algorithm starting with the nodes for Bill and “The Patriot” to see if they occur in the same tree.

## Data Structures

For this project, you will implement a parent-pointer representation of a general tree for an equivalence relation (Shaffer, pp.172-176). Specific requirements for your implementation are:

- The general tree must use an array-based implementation that can be dynamically resized.
- For the purposes of testing you must ensure that the following are done.
  - Each new object added to the relation should be assigned a unique integer starting with zero that is its index in the array for the tree.
  - The UNION of two trees should be implemented as described below in the description of the add command
  - The roots of the trees in the forest should have a parent pointer value of -1.
  - You should be able to print the contents of the general tree as a list of pairs, each of which is the index of the object, and the index of its parent (with the root having parent pointer -1).

Your design must make appropriate use of classes. The specification may imply the existence of additional classes besides those involved in the implementation of the general tree. Data members of classes must be private.

STL containers classes may be used to implement this project. In particular, you may use an STL vector instead of your own dynamic array implementation.

If an error occurs during the parsing of the input file, there's an error in your code. However, your program should still attempt to recover, by “flushing” the current input line and proceeding to the next input line.

## Input

Your program **must** read commands from a file named `RecIn.txt` — use of another input file name will almost certainly result in a score of 0. As before, lines beginning with a semicolon ( `;` ) character are comments; your program will ignore comments. An arbitrary number of comment lines may occur in the input file.

Each non-comment line of the commands file will specify one of the commands described below. Each command line consists of a sequence of “tokens” which will be separated by single tab characters. A newline character will immediately follow the final “token” on each line. Commands are case-sensitive.

The mandatory supported commands are:

`likes <string> <string>`

Add the given pair to the forest representing the likes relation. This will result in a UNION in the general tree; in which case, the root of the tree for the first argument should be made the parent of the root of the second tree. Following this requirement is significant for the display of the tree. The arguments are strings of characters other than tabs and newlines.

`query <string> <string>`

Test if two arguments are related. If so, then print “yes”, otherwise print “no”. If one of the strings does not occur in the relation, print “no”. The arguments have the same format as for the “likes” command.

`display`

Display the parent pointer array of the general tree implementing the relation. The number of each object, along with its parent should be printed on each line as shown below. If the object is the root of its tree, then its parent should be -1.

Your program should exit at end of file.

You may assume that all commands in given input files will have correct syntax.

## Output

Your program **must** write its output to a file named `RecOut.txt` — use of another output file name will undoubtedly result in a score of 0. Since your output for this assignment will be graded automatically, you must adhere to the specified format exactly. A sample output file is given at the end of this specification.

The first two lines should contain your name, course and project title, followed by separator line, as shown below. The remainder will be responses to commands in the file. In particular, you must echo the commands (but not the comments) to the output file, labeled as shown. The output, if any, from processing each command must be written to the output file immediately after the command echo, as shown. Each command and its accompanying output must be delimited in some way (e.g., the lines of hyphens used here).

The output file at the end of this specification was produced by Keller's solution, executed on the given sample command file. The separator lines may use any symbol and be of any length you like.

## Submission

You will submit this assignment to the Curator System (read the *Student Guide*), and it will be graded automatically. Instructions for submitting, and a description of how the grading is done, are contained in the *Student Guide*.

Note that the automated grading system requires that your program be submitted as a single source file. For this project you should develop your implementation using separate compilation, and then manually concatenate the code into a single .cpp file, being careful to get the order right and remove obsolete #include directives.

You will be allowed up to five submissions for this assignment. Use them wisely. Test your program thoroughly before submitting it. If you do not get a perfect score, analyze the problem carefully and test your fix before submitting again. The highest score you achieve will be counted.

The Student Guide can be found at: <http://ei.cs.vt.edu/~eags/Curator.html>

The submission client can be found at: <http://spasm.cs.vt.edu:8080/curator/>

## Evaluation

Your submitted program will be assigned a score based upon the runtime testing performed by the Curator System. After that, your program will be given a brief evaluation by one of the GTAs, who will consider:

- whether your implementation makes appropriate use of data structures (in particular, the use of a well-designed General Tree interface), and
- whether your design shows a good object-oriented decomposition of the given problem, and
- whether the internal documentation of your code is acceptable.

This evaluation will produce a deduction (possibly zero, of course) that will be applied to your runtime testing score to produce your final score for the project.

## Programming Standards

While we won't be carefully evaluating your source code on this assignment for programming style, you should still observe good practice. See the Programming Standards page on the course website for specific requirements that should be observed in this course.

## Pledge

Each of your program submissions must be pledged to conform to the Honor Code requirements for this course. Specifically, you **must** include the following pledge statement in the header comment for your program:

```
//      On my honor:
//
//      - I have not discussed the C++ language code in my program with
//        anyone other than my instructor or the teaching assistants
//        assigned to this course.
//
//      - I have not used C++ language code obtained from another student,
//        or any other unauthorized source, either modified or unmodified.
//
//      - If any C++ language code or documentation used in my program
//        was obtained from another source, such as a text book or course
//        notes, that has been clearly noted with a proper citation in
//        the comments of my program.
//
//      - I have not designed this program in such a way as to defeat or
//        interfere with the normal operation of the Curator System.
//
//      <Student Name>
```

**Failure to include this pledge in a submission is a violation of the Honor Code.**

**Sample Input File**

```

; test file for recommender system
display
;
likes    Bill    "Braveheart"
;
query    Bill    "Braveheart"
query    Ben     "Braveheart"
display
likes    Ben     "Braveheart"
likes    Ben     "The Patriot"
likes    Lesley  "The Patriot"
likes    Lesley  "When Harry met Sally"
likes    Jake    "Rescue Rangers"
likes    Nate    "Rescue Rangers"
likes    Nate    "Purple Fury: Barney's Revenge"
display
query    Bill    "When Harry met Sally"
query    Bill    "Rescue Rangers"
query    Ben     "Purple Fury: Barney's Revenge"
query    Lesley  "Braveheart"
query    Nate    "Braveheart"
display

```

**Sample Output File**

```

-----
Programmer: Ben Keller
CS 2704, Recommender Project
-----
Display:
Empty tree
-----
Input: Bill likes "Braveheart"
-----
Query: does Bill like "Braveheart"?
yes
-----
Query: does Ben like "Braveheart"?
no
-----
Display:
  0  -1
  1   0
-----
Input: Ben likes "Braveheart"
-----
Input: Ben likes "The Patriot"
-----
Input: Lesley likes "The Patriot"
-----
Input: Lesley likes "When Harry met Sally"
-----
Input: Jake likes "Rescue Rangers"
-----
Input: Nate likes "Rescue Rangers"
-----
Input: Nate likes "Purple Fury: Barney's Revenge"
-----
Display:
  0  2
  1  0
  2  4
  3  2
  4 -1
  5  4

```

```
6 8
7 6
8 -1
9 8
```

```
-----
Query: does Bill like "When Harry met Sally"?
yes
```

```
-----
Query: does Bill like "Rescue Rangers"?
no
```

```
-----
Query: does Ben like "Purple Fury: Barney's Revenge"?
no
```

```
-----
Query: does Lesley like "Braveheart"?
yes
```

```
-----
Query: does Nate like "Braveheart"?
no
```

```
-----
Display:
```

```
0 2
1 0
2 4
3 2
4 -1
5 4
6 8
7 6
8 -1
9 8
```