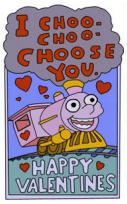
CSE 303, Spring 2009
Homework 4: I Choo-Choo-Choose You (pointers), 50 points
Due Saturday, May 2, 2009, 11:30 PM



This assignment focuses on memory allocation, pointers, and linked data structures in C programs. It is inspired by a similar assignment given by Stuart Reges at Stanford.

Turn in a file named `valentine.c` from the Homework section of the course web site. You will also want the various input and support files from the web site.

The students of Springfield Elementary School are giving out their valentines for Valentine's Day. But the number of valentines is limited; each student can give out only a few. Your task is to help the students figure out to whom they should give their valentines. You will do this using a set of linked lists.

Every student has a set of "scores" from 0-99 that represent how much they like each other student. Any score of 80 or greater means the student likes the classmate enough to potentially give them a valentine. Scores are asymmetrical; Ralph might give Lisa a score of 99, but Lisa might give Ralph a score of 45. This might lead to Ralph giving Lisa a valentine but her not giving one to him. We won't worry about gender; any student can give a valentine to any other.

In this program you will read input from standard-in (redirected in from a file) representing the set of students and their scores for each other. The first line of the input contains two integers representing the total number of students and the number of valentines available to each student. After this is a series of lines, each of which describes a student's score for one other student. The lines appear in no particular order. Students are represented by numbers beginning with 1, rather than by name. The file ends with a line containing 0 0 0 to indicate the end of the input. For example:

```
5 2
1 4 82
4 5 95
3 1 84
1 3 99
2 1 90
2 5 85
1 2 56
4 2 95
3 4 34
4 3 88
2 3 83
0 0 0
```

Figure 1 `input1.txt` file

In the above input data, the first line indicates that there are 5 students and that each one can give out up to 2 valentines. (Not every student will give out all their valentines. If a given student doesn't like 2 other students with an interest level of 80 or more, that student would not give out all 2 of the possible valentines.) The rest of the lines indicate pairs' scores for each other. For example, the line `1 4 82` indicates that student 1 likes student 4 with a score of 82. The file does not necessarily contain a line for every possible pair of students in both directions; any missing scores are assumed to be 0.

It might seem that this program would be best to implement using a 2-D array, where element [*i*][*j*] holds student (*i*+1)'s score toward student (*j*+1) in the data set. (The student numbers begin with 1, not with 0 the way array indexes do, so we need to adjust by 1.) No student gives a valentine to him/herself, so those elements get a score of 0. For example:

	0	*1*	*2*	*3*	*4*
0	0	56	99	82	0
1	90	0	83	0	85
2	84	0	0	34	0
3	0	95	88	0	95
4	0	0	0	0	0

Table 1 Sparse 2-D matrix view of valentine data

It won't be practical to use such a structure, though, because it takes up too much memory and is not dynamic. Fortunately the data in this problem is actually a *sparse matrix*, which means that most of its elements are 0 (or any score less than 80) and therefore do not need to be stored. Thus, you will instead use an array of linked lists to save space.

Linked List Structure:

What you will do instead of making a large 2-D array is to create a linked list for each student that stores only that student's relevant scores (scores of 80 or more). You will still have an overall 1-D array where element [*i*] stores a pointer to the front of student *i*'s linked list. The previous 2-D array could be represented by the following array of linked lists:

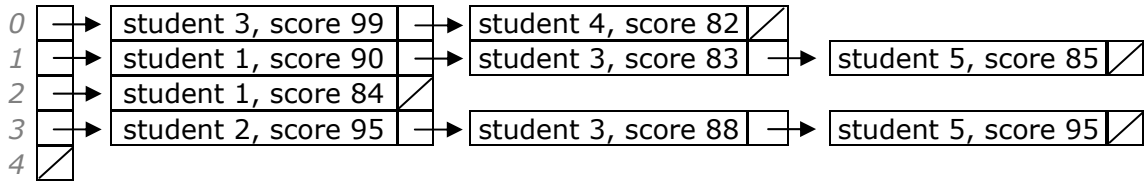


Table 2 One possible linked list view of valentine data

Note that all of the scores under 80 have been discarded. Also note that student #5 (index 4) doesn't know any of the other students (there are no lines in the input file that begin with this student's number), so that linked list is `NULL`.

Each student can give out only 2 valentines, and your goal is to figure out who are the two most desirable people to give each student's valentines to. To do this, your list actually should not match the above diagram, but it should instead by built in sorted non-decreasing order by score (in the case of a tie score, place the later input line earlier in the list):

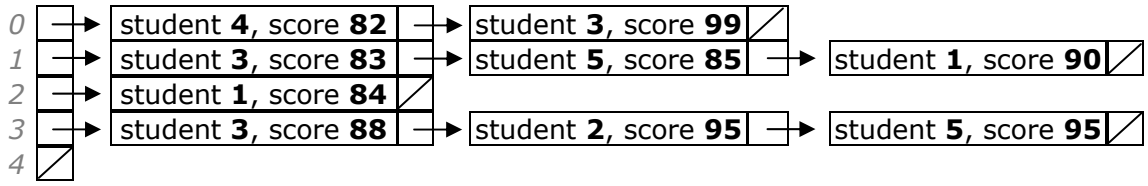


Figure 2 Sorted linked list

Initially your program should read the input data and display the contents of the linked list above. As you read each line of input, you should insert it into the appropriate linked list in the proper location to maintain sorted order.

Next you should **prune** (trim) the linked lists so that none contains more than the maximum number of allowed valentines. Because you have built your linked lists in sorted order, the nodes to discard occur at the front and the nodes to retain occur at the end. The following would be the array of pruned linked lists:

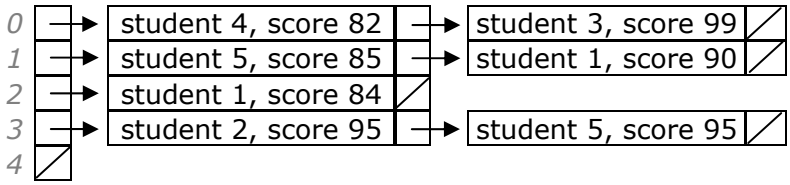


Figure 3 Pruned linked list

Your output should match the following format exactly (excluding whitespace at the end of lines). The last two lines show the total valentines given by all students and the sum of the scores in the matches that led to valentines being given.

```
$ ./valentine < input1.txt
Student    Scores
      1         4 (82)         3 (99)
      2         3 (83)         5 (85)         1 (90)
      3         1 (84)
      4         3 (88)         2 (95)         5 (95)

Student    Valentines Given
      1         4 (82)         3 (99)
      2         5 (85)         1 (90)
      3         1 (84)
      4         2 (95)         5 (95)

Number of students = 5
Total number of valentines given = 7
Sum of all scores = 630
```

Figure 4 Expected output when run with `input1.txt`

The Student number is right-aligned in a column exactly 7 spaces wide. This is followed by 3 spaces. Each valentine listed in the Scores and Valentines Given columns begins with the student number, right-aligned in a column exactly 6 spaces wide. This is followed by the score in parentheses. If there is a next valentine, it will have 3 spaces, then the next student number in a 6-space wide field, and so on. You do not have to worry about the possibility that the output will be too wide to fit on a single line in the terminal window.

You may assume that all input to your program will be valid in the format shown. The first line will always be there, followed by 0 or more lines of score data, then the 0 0 0 line. All student numbers on all lines will be between 1 and the number of students in the class inclusive. All scores listed will be between 0 and 99 inclusive. No student will have a line of data matching against him/herself, and no two lines will have the same pair of student numbers in the same order.

You may assume that the number of lines of data that will appear before 0 0 0 is less than 10% of the stack or heap memory space available to your program. But do not make any assumptions beyond this; it could be very few lines, none at all, or a very large number. Several input/output files are posted to the course web site to help you verify your program.

Development Strategy and Hints:

We suggest that you solve this assignment in stages, verifying that your code works properly after each stage. Because pointers are hard, you will want to quickly write code that can print out the contents of a linked list, so that you can tell what is being stored in each of your lists.

We strongly suggest that you begin by creating an array of unsorted linked lists rather than sorted ones, because this is easier. When you read each line of input, put its data into the appropriate linked list anywhere you like. This will not give you the correct output, but it will allow you to test your code for building the linked lists and outputting their contents.

Warning: Your program is going to have bugs, and it's going to crash. You will get "segfault" errors and other craziness. To help deal with this, use `gdb`, `ddd`, or another debugging tool along with your own `printf` statements to figure out which lines of code are causing problems. Even if you are too stubborn or lazy to learn how to use the debugger, if you get a segfault, at least run the program again in `gdb` to find out the line number on which the crash occurred.

You can verify the correctness of your output using `diff`. You may want to use `-b` or `-w` to ignore blank spaces.

Grading:

Over half of the points will come from the behavioral ("external") correctness and output of your program when run with various test cases. Getting correct output for the preceding `input1.txt` test case does not guarantee full credit. You should test your program using the other provided input files as well as performing your own testing.

A significant amount of points will also come from the style, design, and "internal correctness" of your code. You should avoid redundancy and overly complex as much as possible in the limited context of the shell programming language. You should minimize the use of global variables. The only acceptable global variables are primitive constants declared with the `const` keyword so that their values cannot be changed. You should use functions appropriately both to split up your program to indicate its structure, and also to capture repeated code that would otherwise be redundant.

For full credit, you must store your data in an array (or list) of linked lists. You may use libraries `stdio.h`, `stdlib.h`, and `stdbool.h` on this assignment, but not pre-written data structures or other libraries without instructor permission.

Part of your grade will be based on your appropriate allocation of memory. You should show an understanding of when to allocate data on the stack vs. on the heap, appropriately use pointers, `malloc`/`calloc`, and `free`. You must explicitly free any nodes that you remove from a linked list. You do not need to free the overall lists or the array containing them at the end of your program. (Freeing your memory is mostly just academic on this program, but please do it anyway.)

Format your code nicely with proper indentation, spacing, and clear variable names. Place a descriptive comment heading at the top of your program, as well as a descriptive comment header atop each function, and brief comments throughout your code inside methods explaining what each major section of code is doing. For reference, our solution is around 150 lines long (95 lines long if you exclude blank lines and comments), though you do not need to match this exactly to get full credit; it is just a rough guideline. Your program should produce no errors or warnings from `gcc -Wall`.

Different computers can behave differently with the same C program; for full credit, your program must be able to compile and run successfully either on `attu`, or on the basement lab computers, or on a fresh Ubuntu installation (with `gcc` and the standard C packages installed).