CS211 Fall 2018
Programming Assignment II

David Menendez

Due: Tuesday, October 16, 3:00 AM

This assignment is designed to give you some initial experience with programming in C, as well as compiling, running, and debugging. Your task is to write six C programs.

Section 1 describes the six programs, section 2 describes how your project will be graded, and section 3 describes how to structure and submit your project. Please read the entire assignment description before beginning the assignment.

Note that the assignment is due at 3:00 AM. **Late submissions will not be accepted or graded.** You are strongly encouraged not to work until the last minute. Plan to submit your assignment no later than Monday, October 15.

Advice You are responsible for knowing how to use your tools. You are advised to download and unpack the auto-grader (see section 3.4) first, then create your `src` directory inside `pa1`. Begin work on a program by creating its subdirectory, makefile, and a stub source file containing a `main` function that immediately returns `EXIT_SUCCESS` (see section 3.1). Then run the auto-grader, to make sure that the auto-grader can use your makefile.

As you develop your program, run the auto-grader periodically. This will tell you which aspects of the program you still need to work on. If you like, create user tests (see section 3.5) to cover any additional cases you want to test.

Several programs involve reading different commands. Not all test cases require all commands to be implemented, so you may begin testing before you have completed all commands. This may help you discover bugs earlier.

Once you complete a program, create your `pa1.tar` submission (see section 3.3) and submit to Sakai. This will ensure that you understand how to submit, and will give you time to clarify any instructions. You may submit to Sakai an unlimited number of times. If you have not submitted by the deadline, no later submissions will be accepted.

1 Program descriptions

You will write six programs for this project. Except where explicitly noted, your programs may assume that their inputs are properly formatted. However, your programs should be robust. Your program should not assume that it has received the proper number of arguments, for example, but should check and report an error where appropriate. Except when specified, you may report errors in any manner you prefer.

Programs should always terminate with exit code 0 (that is, return `EXIT_SUCCESS` or 0 from `main`).

1.1 llist: Linked list

Write a program `llist` that maintains and manipulates a sorted linked list of integers according to instructions received from standard input. The linked list is maintained in order, meaning that each integer will be larger than the one preceeding it.

`llist` maintains a linked list containing zero or more integers, ordered from least to greatest. `llist` will need to allocate space for new nodes as they are created using `malloc`; any allocated space should be deallocated using `free` before `llist` terminates.

It supports two operations:

insert *n* Adds an integer *n* to the list. If *n* is already present in the list, it does nothing. The instruction format is an `i` followed by a space and an integer *n*.

delete *n* Removes an integer *n* from the list. If *n* is not present in the list, it does nothing. The instruction format is a `d` followed by a space and an integer *n*.

After each command, `llist` will print the length of the list followed by the contents of the list, in order from first (least) to last (greatest).

Your program will halt once it reaches the end of standard input.

Input format Each line of the input contains an instruction. Each line begins with a letter (either "i" or "d"), followed by a space, and then an integer. A line beginning with "i" indicates that the integer should be inserted into the list. A line beginning with "d" indicates that the integer should be deleted from the list.

Your program will not be tested with invalid input. You may choose to have `llist` terminate in response to invalid input.

Output format After performing each instruction, `llist` will print a single line of text containing the length of the list, a colon, and the elements of the list in order, all separated by spaces.

Usage Because `llist` reads from standard input, you may test it by entering inputs line by line from the terminal. You may use any invalid input, such as `#`, to terminate your session.

```
$ ./llist
i 5
1 : 5
d 3
1 : 5
i 3
2 : 3 5
i 500
3 : 3 5 500
d 5
2 : 3 500
#
```

Alternatively, you may use the shell to provide a file to `llist` using input redirection, e.g.,

```
$ ./llist < tests/test.01.txt
1 : 10
2 : 10 11
3 : 9 10 11
2 : 9 10
```

1.2 mexp: Matrix exponentiation

Write a program `mexp` that multiplies a square matrix by itself a specified number of times. `mexp` takes a single argument, which is the path to a file containing a square ($k \times k$) matrix M and a non-negative exponent n. It computes M^n and prints the result.

Note that the size of the matrix is not known statically. You must use `malloc` to allocate space for the matrix once you obtain its size from the input file.

To compute M^n, it is sufficient to multiply M by itself $n-1$ times. That is, $M^3 = M \times M \times M$. Naturally, a different strategy is needed for M^0.

Input format The first line of the input file contains an integer k. This indicates the size of the matrix M, which has k rows and k columns.

The next k lines in the input file contain k integers. These indicate the content of M. Each line corresponds to a row, beginning with the first (top) row.

The final line contains an integer n. This indicates the number of times M will be multiplied by itself. n is guaranteed to be non-negative, but it may be 0.

For example, an input file `file.txt` containing

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

indicates that `mexp` must compute

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}^2 .$$

Output format The output of `mexp` is the computed matrix M^n. Each row of M^n is printed on a separate line, beginning with the first (top) row. The items within a row are separated by spaces.

Using `file.txt` from above,

```
$ ./mexp file1.txt
30 36 42
66 81 96
102 126 150
```

1.3 stddev: Standard deviation

Write a program `stddev` which computes the mean and standard deviation of a sequence of numbers. The program will take no arguments, and will read the sequence of numbers from standard input.

Background Let X be a set of numbers. The *mean* of X is defined as

$$\overline{X} = \frac{\sum_{x \in X} x}{|X|}, \tag{1}$$

where $|X|$ is the number of elements in X. That is, $\overline{X}$ is the sum of the elements in X divided by the number of elements.

The *standard deviation* of X is the square root of the mean squared difference between each element of X and $\overline{X}$. That is,

$$s = \sqrt{\frac{\sum_{x \in X} (x - \overline{X})^2}{|X|}} \tag{2}$$

Note that the mean and standard deviation are not defined if $X = \emptyset$. Your program must detect this condition, and avoid dividing by zero.

Input The input to `stddev` is a sequence of zero or more decimal numbers, separated by whitespace (e.g., newlines). The input is terminated when the input ends.

Your program will not be tested with invalud input. You may have `stddev` terminate after reading invalid input.

For example,

```
 82.5
 1000.6699
 10
 11.11
-45
 #
```

Output Your program will output two lines after it has completed its computation, giving the mean and standard deviation of its input. To minimize discrepencies caused by rounding error, the output will be rounded to an integer using the `%.0f` formatting code.

The output will have this form,

```
mean: 105
stddev: 13
```

Your program must use lower-case letters, with a single space after the colon.

In the case where the input contains no numbers, your program will instead write `no data`.

Usage Because `stddev` reads from standard input, you may provide input numbers by typing into the terminal after invoking it.

```
$ ./stddev
45
8
3.2
16
```

```
#
mean: 18
stddev: 16
$ ./stddev
#
no data
```

Alternately, you may instruct the shell to send a file to `stddev` using input redirection, e.g.,

```
$ ./stddev < some_file.txt
mean: 10
stddev: 8
```

Notes To compute the square root, you may use the `sqrt` function provided by `math.h`. Using `math.h` requires linking against the math library, so your makefile must include `-lm` as an option for `gcc`.

Your program will not know in advance how large its input will be, and will not be able to read its input more than once. You will need some way of storing a sequence of numbers whose size is not known in advance.

1.4 bst: Binary search trees

Write a program `bst` that manipulates binary search trees. It will receive commands from standard input, and print resposes to those commands to standard output.

A binary search tree is a binary tree that stores integer values in its interior nodes. The value for a particular node is greater than every value stored its left sub-tree and less than every value stored in its right sub-tree. The tree will not contain any value more than once. `bst` will need to allocate space for new nodes as they are created using `malloc`; any allocated space should be deallocated using `free` before `bst` terminates.

This portion of the assignment has two parts.

Part 1 In this part, you will implement `bst` with three commands:

insert *n* Adds a value to the tree, if not already present. The new node will always be added as the child of an existing node, or as the root. No existing node will change or move as as result of inserting an item. If *n* is already present in the tree, `bst` will print `duplicate`. Otherwise, it will print `inserted`. The instruction format is an `i` followed by a decimal integer *n*.

search *n* Searches the tree for a value *n*. If *n* is present, `bst` will print `present`. Otherwise, it will print `absent`. The instruction format is an `s` followed by a space and an integer *n*.

print Prints the current tree structure, using the format in section 1.4.1.

Part 2 In this part, you will implement `bst` with an additional fourth command:

delete *n* Removes a value from the tree. See section 1.4.2 for further discussion of deleting nodes. If *n* is not present, print `absent`. Otherwise, print `deleted`. The instruction format is a `d` followed by a space and an integer *n*.

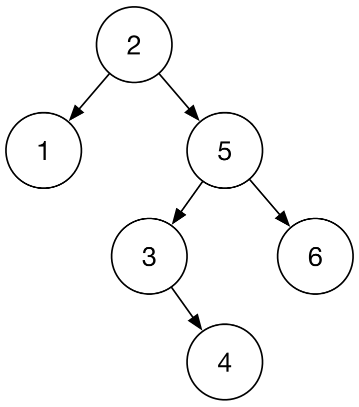


Figure 1: A binary search tree containing six nodes

Input format The input will be a series of lines, each beginning with a command character (`i`, `s`, `p`, or `d`), possibly followed by a decimal integer. When the input ends, the program should terminate.

Your program will not be tested with invalid input. A line that cannot be interpreted may be treated as the end of the input.

Output format The output will be a series of lines, each in response to an input command. Most commands will respond with a word, aside from `p`. The format for printing is described in section 1.4.1.

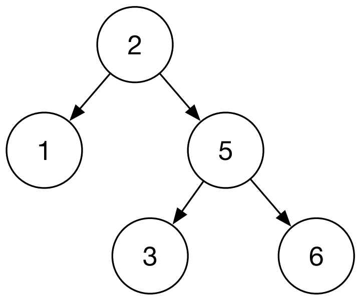
Usage

```
$ ./bst
i 1
inserted
i 2
inserted
i 1
duplicate
s 3
absent
p
(1(2))
#
```

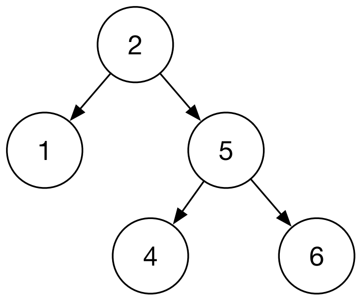
1.4.1 Printing nodes

An empty tree (that is, `NULL`) is printed as an empty string. A node is printed as a `(`, followed by the left sub-tree, the item for that node, the right subtree, and `)`, without spaces.

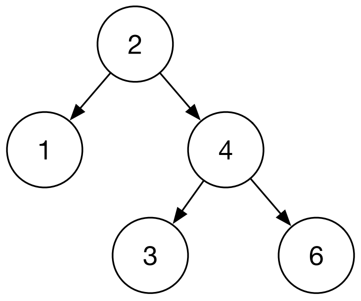
For example, the output corresponding to fig. 1 is `((1)2((3(4))5(6)))`.



(a) Deleted 4



(b) Deleted 3



(c) Deleted 5

Figure 2: The result of deleting different values from the tree in fig. 1

1.4.2 Deleting nodes

There are several strategies for deleting nodes in a binary tree. If a node has no children, it can simply be removed. That is, the pointer to it can be changed to a `NULL` pointer. Figure 2a shows the result of deleting 4 from the tree in fig. 1.

If a node has one child, it can be replaced by that child. Figure 2b shows the result of deleting 3 from the tree in fig. 1. Note that node 4 is now the child of node 5.

If a node has two children, its value will be changed to the maximum element in its left subtree. The node which previously contained that value will then be deleted. Figure 2c shows the result of deleting 5 from the tree in fig. 1. Note that the node that previously held 5 has been relabeled 4, and that the previous node 4 has been deleted.

Note that the value being deleted may be on the root node.

1.5 life: Conway's Game of Life

Write a program `life` that plays a specified number of steps in the Game of Life, given an intial configuration. `life` takes two arguments: an integer N indicating how many steps it should perform, and the name of a file containing the initial configuration.

If N is not a positive integer, or if the file does not exist or cannot be opened, `life` should report an error and terminate.

Background The Game of Life is a cellular automaton created by John Conway. It is an example of a zero-player game, where each step is completely determined by the current state of the board.

Life is played on an infinite, two-dimensional grid of square *cells*. Each cell has eight *neighbors*, which are the cells that share a side or a corner with it. At any time, each cell is either populated or unpopulated. If a cell is populated at time t but has fewer than two populated neighbors or more than three populated neighbors, it will be unpopulated at time $t + 1$. If a cell is unpopulated at time t but has exactly three populated neighbors, it will be populated at time $t + 1$.

The idea is that populations require a certain size in order to thrive, and can expand if they are large enough, but cannot become so dense that they overtax their local resources.

An infinite grid is impractical to simulate, so we will use a grid that wraps around, so that the top row and bottom row are neighbors, as are the left and right sides.[1]

Your program will read an initial board from a file and output the configuration that board would reach after N steps.

Input format The input file has two parts. The first line contains two integers, h and w, indicating the height and width of the board, respectively.

The following h lines describe the board. Each row is given by a line containing w characters, indicating the status of a cell. A populated cell is indicated by an asterisk (`*`) and an unpopulated cell by a period (`.`).

For example, an input file `board.txt` might contain

```
5 5
.....
..*..
..*..
..*..
.....
```

Your program will not be tested with invalid input.

Output format The output of `life` is another board configuration, in the same format as the input.

Usage

```
$ ./life 1 board.txt
.....
.....
.***.
.....
.....
$ ./life 2 board.txt
.....
..*..
..*..
```

[1]That is, the grid has the topology of a *torus*.

```
..*..
.....
```

1.6 queens: $(N+1)$-queens

Write a program `queens` that implements part of the search process for solving the N-queens problem. `queens` has one required argument and two optional arguments. The required argument will always occur last: it is the name of a file containing a chessboard.

The two optional arguments may occur individually or together in any order. They are `+2`, which indicates that `queens` should report whether two queens may be placed on the board (Part 2), and `-w`, which allows for the presence of warrior queens (Part 3).

This portion of the assignment has three parts, two of which are required. The third part will be counted as extra credit.

Background A chessboard is an 8×8 grid of squares, each of which may contain up to one chess piece.

The *queen* is the most powerful piece in chess, capable of moving any number of squares vertically, horizontally, or diagonally, provided that it does not change direction during the move.

The N-queens problem asks how many ways N queens can be arranged on a board such that no queen can take another. That is, no queen can be moved to a square occupied by another queen in a single move.

Part 1 Your program will be given a chessboard with some number of queens already placed. Your program must first determine whether any queen can take any other queen. If so, it will print `Invalid`. If not, it will determine, for each empty square, whether a queen placed there could be taken by any other queen. It will print a new chessboard with these available squares marked with `q`.

Part 2 When `+2` is given, your program will further determine if there is any way to place *two* additional queens on the board such that no queen can take any other queen. If this is possible, `queens` will print `Two or more`. If only one queen may be placed, `queens` will print `One`. If no queens may be placed, `queens` will print `Zero`.

Part 3 (Extra credit) A *warrior queen* is a queen that fights on horseback, and thus may move like a queen or like a *knight*. The knight moves in an L-shape: two squares horizontally and one square vertically, or two squares vertically and one square horizontally. Thus, a warrior queen may reach up to eight squares that a regular queen in the same position could not.

When `-w` is given, `queens` will consider boards that may contain warrior queens and distinguish between squares where a warrior queen may be placed (marked `w`) and squares where only a regular queen may be placed (marked `q`). Additionally, `queens` will accept chessboards containing warrior queens.

Input format The input will contain a chessboard, specified as eight lines containing eight characters. A `.` indicates an empty space, and a `Q` indicates a queen.

A `W` indicates a warrior queen. Behavior is unspecified if the chessboard contains a warrior queen, but the `-w` option has not been given.

Your program will not be tested with incorrectly formatted chessboards.

Output format If any queens on the board may take another, `queens` will print `Invalid` and terminate. Otherwise, `queens` will print a chessboard similar to the input, but with additional symbols `q` and `w` to indicate squares where a queen or warrior queen may be placed. The mark `w` will only be used if the `-w` option is given.

Next, if the `+2` option is given, `queens` will print `Zero`, `One`, or `Two or more`, depending on how many additional queens may be safely placed on the board.

Usage Assume the file `board.txt` contains the following chessboard:

```
........
........
....Q...
......Q.
........
..Q.....
........
........
```

Then,

```
$ ./queens board.txt
qq...q..
qq.....q
....Q...
......Q.
q.......
..Q.....
.....q.q
.q.q.q.q
$ ./queens +2 -w board.txt
ww...q..
ww.....q
....Q...
......Q.
q.......
..Q.....
.....w.w
.q.q.w.w
Two or more
```

2 Grading

Each program will be awarded points during testing, for a total of 150 possible points for the assignment. Table 1 gives the points available for each part using the test cases provided with the assignment and the full set of test cases used for final scoring. An additional 20 extra credit points are available for completing Part 3 of `queens`. Extra credit points are not included in the score for this project, but will be kept separately until the final grades for the semester are determined.

Table 1: Points available per program

Program	Provided	Final
`llist`	20	40
`mexp`	20	40
`stddev`	20	40
`bst`		
Part 1	20	40
Part 2	10	20
`life`	20	40
`queens`		
Part 1	20	40
Part 2	20	40
`total`	150	300

1. The auto-grader will award points for each program, according to how many test cases the program completes successfully.
2. If the makefile for the program does not include `-Wall -Werror -fsanitize=address`, its score is multiplied by 0.8.

The auto-grader provided for students includes several test cases, but additional test cases will be used during grading. Make sure that your programs meet the specifications given, even if no test case explicitly checks it. It is advisable to perform additional tests of your own devising. Section 3.5 describes how to use the auto-grader to perform additional tests that you create. Note that these additional tests will not contribute to your final score.

2.1 Academic integrity

You must submit your own work. You should not copy or even see code for this project written by anyone else, nor should you look at code written for other classes. We will be using state of the art plagiarism detectors. Projects which the detectors deem similar will be reported to the Office of Student Conduct.

Do not post your code on-line or anywhere publically readable. If another student copies your code and submits it, both of you will be reported.

3 Submission

Your solution to the assignment will be submitted through Sakai. You will submit a Tar archive file containing the source code and makefiles for your project. Your archive should not include any compiled code or object files.

The remainder of this section describes the directory structure, the requirements for your makefiles, how to create the archive, how to use the provided auto-grader, and how to create your own test files to supplement the auto-grader.

3.1 Directory structure

Your project should be stored in a directory named `src`, which will contain three sub-directories. Each subdirectory will have the name of a particular program, and contain (1) a makefile, and (2) any source files needed to compile your program. Typically, you will provide a single C file named for the program. That is, the source code for the program `llist` would be a file `llist.c`, located in the directory `src/llist`.

This diagram shows the layout of a typical project:

```
src
 +- llist
 |   +- Makefile
 |   +- llist.c
 +- bst
 |   +- Makefile
 |   +- bst.c
 +- mexp
 |   +- Makefile
 |   +- mexp.c
 +- stddev
 |   +- Makefile
 |   +- stddev.c
 +- life
 |   +- Makefile
 |   +- life.c
 +- queens
     +- Makefile
     +- queens.c
```

If you are providing your own tests, as described in section 3.5, they will be in a directory named `test` inside the program directory. For example,

```
src
+- llist
|   +- Makefile
|   +- llist.c
|   +- tests
|       +- test.01.txt
|       +- test.01.ref.txt
...
```

3.2 Makefiles

We will use `make` to manage compilation. Each program directory will contain a file named `Makefile` that describes at least two targets. The first target must compile the program. An additional target, `clean`, must delete any files created when compiling the program (typically just the compiled program).

A typical makefile for the program `llist` would be:

```
llist: llist.c
	gcc -g -Wall -Werror -fsanitize=address -o llist llist.c

clean:
	rm -f llist
```

Note that the command for compiling `llist` uses GCC warnings and the address sanitizer. You will points if your makefile does not include these.

Note also that the command for compiling `llist` includes `-g`. This will enable use of the debugger, and may help AddressSanitizer produce more helpful error messages. Use of this option is not required.

Note that the makefile format requires that lines be indented using a single tab, not spaces. Be aware that copying and pasting text from this document will "helpfully" convert the indentation to spaces. You will need to replace them with tabs (literal tab characters), or simply type the makefile yourself. You are advised to use make when compiling your program, as this will ensure (1) that your makefile works, and (2) that you are testing your program with the same compiler options that the auto-grader will use.

3.3 Creating the archive

We will use `tar` to create the archive file. To create the archive, first ensure that your `src` directory contains only the source code and makefiles needed to compile your project. Any compiled programs, object files, or other additional files must be moved or removed.

Next, move to the directory containing `src` and execute this command:

```
tar czvf pa2.tar src
```

`tar` will create a file `pa2.tar` that contains all files in the directory `src`. This file can now be submitted through Sakai.

To verify that the archive contains the necessary files, you can print a list of the files contained in the archive with this command:

```
tar tf pa2.tar
```

You should also use the auto-grader to confirm that your archive is correctly structured.

It is okay to include any user tests you may have created (see section 3.5), but these will not contribute to your grade.

3.4 Using the auto-grader

We have provided a tool for checking the correctness of your project. The auto-grader will compile your programs and execute them several times with different arguments, comparing the results against the expected results.

Setup The auto-grader is distributed as an archive file `pa2_grader.tar`. To unpack the archive, move the archive to a directory and use this command:

```
tar xf pa2_grader.tar
```

This will create a directory `pa2` containing the auto-grader itself, `grader.py`, a library `autograde.py`, and a directory of test cases `data`.

Do not modify any of the files provided by the auto-grader. Doing so may prevent the auto-grader from correctly assessing your program.

You may create your `src` directory inside `pa2`. If you prefer to create `src` outside the `pa2` directory, you will need to provide a path to `grader.py` when invoking the auto-grader (see below).

Usage While in the same directory as `grader.py` and `src`, use this command:

```
python grader.py
```

The auto-grader will compile and execute the programs in the directory `src`, assuming `src` has the structure described in section 3.1.

By default, the auto-grader will attempt to grade all programs. You may also provide one or more specific programs to grade. For example, to grade only `gcd`:

```
python grader.py gcd
```

To obtain usage information, use the `-h` option.

Program output By default, the auto-grader will not print the output from your programs, except for lines that are incorrect. To see all program output, use the `-v` option:

```
python grader.py -v
```

To see program output for all tests, use `-vv`. To see no program output, use `-q`.

Checking your archive We recommended that you use the auto-grader to check an archive before submitting. To do this, use the `-a` option with the archive file name. For example,

```
python grader.py -a pa2.tar
```

This will unpack the archive into a temporary directory, grade the programs, and then delete the temporary directory.

Specifying source directory If your `src` directory is not located in the same directory as `grader.py`, you may specify it using the `-s` option. For example,

```
python grader.py -s ../path/to/src
```

3.5 Providing your own tests

The tests provided with the auto-grader may not check every possible input scenario. If you wish to be certain that your program is correct, you should design and test additional test cases. This may be done by entering the cases manually and running the program yourself, or by using the user test capability of the auto-grader.

To create additional tests for the auto-grader, create a directory named `tests` inside the directory containing your souce code. For each test, you will create a pair of files describing the input to the program and the expected output (the *reference*).

For `llist`, `mexp`, `stddev`, and `bst`, the input and reference files will be named `test.`N`.txt` and `test.`N`.ref.txt`, respectively, where N is a unique identifier.

For `life`, the input and reference files will be named `test.`N`.txt` and `test.`N`.`S`.ref.txt`, respectively, where N is a unique identifier and S is the number of steps that will be passed to `life` as its first argument. Note that multiple reference files may correspond to a single input file.

For `queens`, the input and reference files will be named `test.`M`.`N`.txt` and `test.`M`.`N`.ref.txt`, respectively, where N is a unique identifier, and M identifies the options passed to `queens`: `1` indicates no options, `2` indicates `+2`, `3` indicates `-w`, and `4` indicates `+2 -w`.

The capacity for user tests is provided as a convenience. You are not required to create user tests. The presence or absence of user tests will not directly affect your score.