CS211 Fall 2019
Programming Assignment I

David Menendez

Due: October 8, 2019, 11:00 PM
Hand in by October 9, 2019, 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 seven small C programs.

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

Note that the assignment is due at 11:00 PM, but submissions will be accepted without penalty at late as 3:00 AM the following morning. **Submissions after the grade period 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 October 7.

1 Program descriptions

You will write seven 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.

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

1.1 factors: Reading arguments

Write a program `factor` that prints the prime factors of a specified non-negative integer. The integer will be provided as a command-line argument.

You may assume that the integer will be greater than 1 and less than or equal to the largest signed integer on your platform (that is, you may use an `int`).

Output Your program should print every prime factor of the requested integer, in order, with each factor on its own line. Note that 1 is not considered prime, so it should never be printed.

Note that prime numbers have exactly one prime factor: themselves.

Usage

```
$ ./factor 17
17
$ ./factor 384277608
2
3
11
13
29
$ ./factor 384277609
19603
```

Notes While your program should be efficient, it is acceptable to use fairly simple algorithms for searching for factors and testing primality. An implementation that takes time proportional to the square of the input number should give sufficient performance.

Depending on your design, it may be possible to cut the number of required divisions in half by recalling that 2 is the only even prime.

1.2 warble: String operations I

Write a program `warble` that echoes its arguments, but transforms roughly half of the lower-case letters to upper-case.

When printing a lower-case letter, `warble` should convert to upper-case if the most recently printed letter was lower-case. All other characters should be printed verbatim. After printing its input, `warble` should print a newline.

If `warble` receives more than one argument, it should print a single space between the arguments. If warble recieves no arguments, it should print only the newline.

Usage

```
$ ./warble hello world
hElLo WoRlD
$ ./warble I am    NASA scientist
I aM NASA sCiEnTiSt
$ ./warble "How do you do, fellow kids?"
HoW dO yOu Do, FeLloW kIdS?
$ ./warble "#include <stdio.h>"
#iNcLuDe <StDiO.h>
$ ./warble blank arguments are "" okay
bLaNk ArGuMeNtS aRe  OkAy
```

Notes Recall that the shell will process the users's command before passing arguments to your program. It is not necessary or possible to reconstruct the exact text entered by the user; simply process the arguments as given.

When testing, you may use single and double quotes and backslash escapes to send special characters to your program.

1.3 rle: String operations II

Write a program `rle` that uses a simple method to compress strings. `rle` takes a single argument and looks for repeated characters. Each repeated sequence of a letter or punctuation mark is reduced to a single character plus an integer indicating the number of times it occurs. Thus, "aaa" becomes "a3" and "ab" becomes "a1b1".

If the compressed string is longer than the original string, `rle` must print the original string instead.

If the input string contains digits, `rle` must print "error" and nothing else.

Usage

```
$ ./rle aaaaaa
a6
$ ./rle aaabcccc..a
a3b1c4.2a1
$ ./rle aaabab
aaabab
$ ./rle a1b2
error
```

1.4 list: Linked lists

Write a program `list` that maintains and manipulates a sorted linked list according to instructions received from standard input. The linked list is maintained in order, meaning that the items in the list are stored in increasing numeric order after every operation.

Note that `list` will need to allocate space for new nodes as they are created, using `malloc`; any allocated space should be deallocated using `free` as soon as it is no longer needed.

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

`list` must 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 `list` terminate in response to invalid input.

Output format After performing each instruction, `list` 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 `list` reads from standard input, you may test it by entering inputs line by line from the terminal.

```
$ ./list
i 5
1 : 5
d 3
1 : 5
i 3
2 : 3 5
i 500
3 : 3 5 500
d 5
2 : 3 500
^D
```

To terminate your session, type Control-D at the beginning of the line. (This is indicated here by the sequence `^D`.) This closes the input stream to `list`, as though it had reached the end of a file.

Alternatively, you may use input redirection to send the contents of a file to `list`. For example, assume `list_commands.txt` contains this text:

```
i 10
i 11
i 9
d 11
```

Then we may send this file to `list` as its input like so:

```
$ ./list < list_commands.txt
1 : 10
2 : 10 11
3 : 9 10 11
2 : 9 10
```

1.5 mexp: Matrix manipulation

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
```

1.6 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 was not present, and hence has been inserted, `bst` will print `inserted`. Otherwise, it will print `not 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 1.6.1.

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

delete *n* Removes a value from the tree. See section 1.6.2 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*.

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 1.6.1.

Usage

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

As with `list`, the `^D` here indicates typing Control-D at the start of a line in order to signal the end of file.

1.6.1 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)))`.

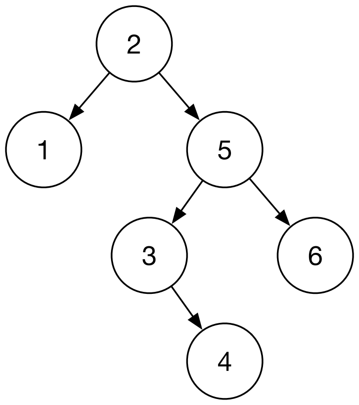
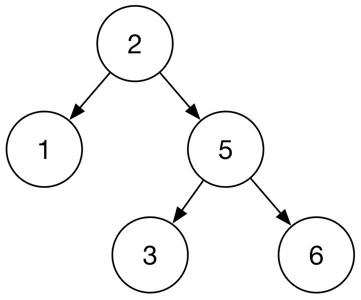
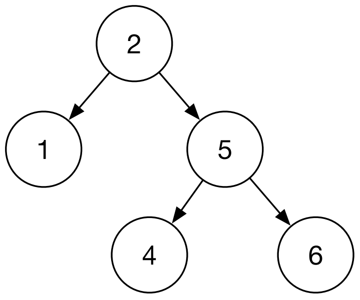


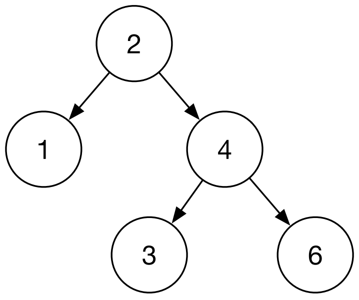
Figure 1: A binary search tree containing six nodes



(a) Deleted 4



(b) Deleted 3



(c) Deleted 5

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

1.6.2 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.

1.7 cell: Cellular automata

Write a program `cell` that simulates two-state, one-dimensional cellular automata given a particular behavior specification. A cellular automaton describes how to update the state of a sequence of cells given a small set of rules. Your program will work with a circular sequence of cells, which may be empty or occupied. The rules determine whether a cell will be occupied at step $t + 1$ based on the state of a cell and its neighbors at step t. When cell is run, it is given the number of cells in the sequence, a description of the rule it should use to find the next states for the cells, and an initial configuration of the sequence.

For example, `cell` might simulate a sequence with 9 cells, where initially only the fifth cell is occupied. We might represent this as `....*....`, where `*` and `.` represent occupied and unoccupied cells, respectively. We might apply the rule that says a cell will be occupied at time $t + 1$ if it or either of its neighboring cells, but not all three, are occupied at time t. (Note that this is a circular sequence, so the first and last cells are considered neighbors.)

Applying this rule, `cell` will determine that the next states will be `...***...`, followed by `..**.**..` and `.*******.`. `cell` will compute a requested number of states, printing each on a separate line. In this example, if we requested six steps, we would get:

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

Operation `cell` will read the configuration of the requested automaton from standard input. The input consists of three or four tokens: two integers, a rule specification string, and an optional initial state string.

The first integer, w, gives the number of cells in the sequence. It will be at least 3.

The second integer, n, gives the number of state transitions to perform. `cell` will print the initial state, followed by the the next n states produced by following the transition rules, for a total of $n + 1$ lines of output.

The rule specification string is a sequence of 8 characters, each of which may be a period (`.`) or an asterisk (`*`). Section 1.7.1 describes how to interpret this string.

If present, the last token will be a sequence of periods and asterisks that gives the initial state of the automaton. If the token contains fewer than w characters, the remaining cells are considered blank. If the token is not present at all, then the initial state will be all blank except for the cell with index $n/2$ (rounded down).

The example given above could be specified with this input:

```
9
6
.******.
....*....
```

The last line could be omitted in this case, since the only occupied cell has index 4, which is 9/2 rounded down.

Inputs that define larger automata can produce more interesting results. For example, given this input:

```
31
15
.*.**.*.
```

your program should produce this output:

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

1.7.1 Rule specification

To determine the state of a cell at time $t + 1$, we consider the states of it and its two neighboring cells at time t. Each of these three cells may be occupied or unoccupied, so there are $2^3 = 8$ possible configuations of a state's neighborhood. A transition rule gives a new state for each possible configuration.

Table 1: A Rule Specification

neighborhood	`...`	`..*`	`.*.`	`.**`	`*..`	`*.*`	`**.`	`***`
next state	`.`	`*`	`*`	`.`	`*`	`.`	`.`	`*`

We can specify a transition rule with a table. For example, table 1 describes a rule where a cell will be occupied in step $t + 1$ if only one or all three of the cells in its neighborhood are occupied in step t.

Because the neighborhood configurations are always given in the same order, only the next states need to be included to specify a rule. Thus, the rule shown in table 1 would be given as `.**.*..*` in the input to `cell`.

2 Grading

Your submission will be awarded up to 100 points, based on how many test cases your programs complete successfully. Each program will receive up to 10 points, except `bst` and `cell`. Parts 1 and 2 of `bst` will be awarded up to 15 points each. `cell` will be awarded up to 20 points.

The auto-grader provided for students includes half of the test cases that will be used during grading. Thus, it will award up to 50 points.

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.

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 `factor` would be a file `factor.c`, located in the directory `src/factor`.

This diagram shows the layout of a typical project:

```
src
 +- factor
 |   +- Makefile
 |   +- factor.c
 +- warble
 |   +- Makefile
 |   +- warble.c
 +- rle
 |   +- Makefile
 |   +- rle.c
 +- llist
 |   +- Makefile
 |   +- llist.c
 +- mexp
 |   +- Makefile
 |   +- mexp.c
 +- cell
 |   +- Makefile
 |   +- cell.c
 +- bst
     +- Makefile
     +- bst.c
```

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 `factor` would be:

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

clean:
        rm factor
```

Note that the command for compiling `factor` uses GCC warnings and the address sanitizer. Your score will be reduced if your makefile does not include these options.

Use of `-g` is recommended, but 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 pa1.tar src
```

`tar` will create a file `pa1.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 pa1.tar
```

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

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 `pa1_grader.tar`. To unpack the archive, move the archive to a directory and use this command:

```
tar xf pa1_grader.tar
```

This will create a directory `pa1` 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 `pa5`. If you prefer to create `src` outside the `pa5` 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 `factor`:

```
python grader.py factor
```

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 for unsuccessful tests, 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 recommend 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 pa1.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
```