CS211 Spring 2018
Programming Assignment I

David Menendez

Due: Wednesday, February 14, at 8:00 PM

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 eight small C programs.

The following sections describe the eight programs, how to structure and submit your project, and the guidelines for grading.

1 Program descriptions

You will write eight 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 0 (that is, return 0 from `main`).

1.1 yell: String operations I (6 points)

Write a program `yell` that is similar to `echo` except that it converts text to all-caps. `yell` takes a single argument, replaces all lower-case letters with upper-case letters, and prints the resulting string.

Usage

```
$ ./yell hello
HELLO
$ ./yell "It's over 9000"
IT'S OVER 9000
```

1.2 rle: String operations II (10 points)

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
```

1.3 anagram: Sorting (10 points)

Write a program `anagram` that sorts the letters in a word, which is useful when searching for anagrams. `anagram` takes a single argument, which is a string containing only lower-case letters, sorts the letters alphabetically, and then prints the sorted letters.

You may use any sorting algorithm you are familiar with, but you must write the sort function yourself. You may not use any sort function provided by a library.

Usage

```
$ ./anagram hello
ehllo
$ ./anagram positivity
iiiopsttvy
$ ./anagram abcdef
abcdef
```

1.4 list: Linked lists (10 points)

Write a program `list` that maintains and manipulates a linked list of integers according to instructions provided in a file. `list` takes a single argument, which is the path to a file containing the instructions. If the file does not exist, your program must print "error" and nothing else.

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

It supports two operations on this list:

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.

Input format Each line of the input file 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.

Output format Normal output for `list` is written on two lines. The first line is an integer indicating the length of the linked list after all instructions are performed. The second line is zero or more integers separated by spaces indicating the content of the linked list.

If the file given on the command line does not exist, `list` should print `error` and nothing else.

Usage Assume three text files. The first, `file1.txt`, is empty. The second, `file2.txt`, contains:

```
i 5
i 2
i 8
i 3
d 2
```

The third, `file3.txt`, contains:

```
d 12
i 10
i 7
i 10
i 5
d 10
```

Assume also that there is no file `file4.txt`.

With these assumptions,

```
$ ./list file1.txt
0

$ ./list file2.txt
3
3 5 8
$ ./list file3.txt
2
5 7
$ ./list file4.txt
error
```

1.5 table: Hash tables (10 points)

Write a program `table` that maintains and manipulates a hash table according to instructions provided in a file. `table` takes a single argument, which is a path to the file containing the instructions.

`table` uses a hash table with 10,000 buckets. It uses a very simple hash function: $h(n) = n \bmod 10,000$. To handle *collisions*—when two different values hash to the same bucket—each bucket should contain a linked list of values. Thus, to determine whether a value n is present in the table, first look up entry $h(n)$ in the bucket array, then search the bucket's list for n.

The operations supported by `table` are:

insert n Add an integer n to the hash table. If n is already present in the table, print `duplicate`. Otherwise, print `inserted`.

search n Check whether n is present in the hash table. If it is, print `present`. Otherwise, print `absent`.

Input format Each line of the input file contains an instruction. The line begins with a single character (either "i" or "s"), followed by a space and then an integer. The characters "i" and "s" indicate insert or search operations, respectively, and the integer represents n.

Output format Normal output for `table` contain the results printed for each instruction, each on its own line. If the argument to `table` is a non-existent file, the output is the word `error` and nothing else.

Usage Assume a text file, `file.txt`, with the following content:

```
i 10
i 5
i 325689
s 5
i 5
s 100
```

With this assumption,

```
$ ./table file.txt
inserted
inserted
inserted
present
duplicate
absent
```

1.6 mexp: Matrix exponentiation (10 points)

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 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, $M^3 = M \times M \times M$.

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
```

1.7 sudoku: Checking Sudoku solutions (16 points)

Write a program `sudoku` that (part 1) checks whether a proposed Sudoku solution is correct and (part 2) checks whether a partially-solved Sudoku puzzle with one unknown square can be solved. `sudoku` takes a single argument, which is the path to a file containing a completed or almost-completed Sudoku puzzle.

A completed Sudoku puzzle is a 9×9 matrix containing the digits 1–9, inclusive. The matrix is divided into nine 3×3 submatrixes, themselves arranged in a 3×3 square. A completed Sudoku puzzle is a *correct solution* if and only if it has the following properties:

- Each digit occurs exactly once in each row (that is, no row contains any digit more than once.
- Each digit occurs exactly once in each column.
- Each digit occurs exactly once in each submatrix.

For example, this completed puzzle is a correct solution:

$$\left[\begin{array}{ccc|ccc|ccc} 5 & 3 & 4 & 6 & 7 & 8 & 9 & 1 & 2 \\ 6 & 7 & 2 & 1 & 9 & 5 & 3 & 4 & 8 \\ 1 & 9 & 8 & 3 & 4 & 2 & 5 & 6 & 7 \\ \hline 8 & 5 & 9 & 7 & 6 & 1 & 4 & 2 & 3 \\ 4 & 2 & 6 & 8 & 5 & 3 & 7 & 9 & 1 \\ 7 & 1 & 3 & 9 & 2 & 4 & 8 & 5 & 6 \\ \hline 9 & 6 & 1 & 5 & 3 & 7 & 2 & 8 & 4 \\ 2 & 8 & 7 & 4 & 1 & 9 & 6 & 3 & 5 \\ 3 & 4 & 5 & 2 & 8 & 6 & 1 & 7 & 9 \end{array}\right]$$

An almost-completed Sudoku puzzle is similar, except that one or two elements are unspecified.

This problem has two parts. For part one (6 points), you must be able to check whether a completed Sudoku puzzle contains a correct solution. For part two (6 points), you must be able to determine whether an almost-completed Sudoku puzzle is solvable. The remaining 4 points are for style. Both parts are submitted as a single program. You will receive partial credit if your program correctly handles part one.

1.7.1 Correctness checking

The input file contains a completed Sudoku puzzle. The puzzle is given on nine lines, each of which contains nine digits without any separation. Note that the digit `0` will not occur. The input file `file1.txt` corresponding to the puzzle above would contain:

```
534678912
672195348
198342567
859761423
426853791
713924856
961537284
287419635
345286179
```

When given such a file, `sudoku` must determine whether the completed puzzle is a correct solution (meaning it satisfies the three properties). If so, it prints `correct`. Otherwise, it prints `incorrect`.

If the input file does not exist, is not readable, or does not follow the format specified here, `sudoku` may print `error`.

Usage

```
$ ./sudoku file1.txt
correct
```

1.7.2 Solvability checking

The input file contains an almost-completed Sudoku puzzle. The format is the same as above—nine lines with nine characters each—except that up to two of the characters may be spaces instead of digits. The spaces indicate that the digit for that element is unspecified.

The input file `file2.txt` contains an almost-completed puzzle with two unknown entries:

```
435269781
682571493
1978345 2
826195347
374682915
951743628
519 26874
248957136
763418259
```

If the input file contains an almost-completed puzzle, `sudoku` must determine whether the puzzle can be solved. If the puzzle has unspecified entries, can they be replaced by a digits such that the completed puzzle is a correct solution? If so, `sudoku` prints `solvable`. Otherwise, it prints `unsolvable`. If the puzzle contains no unspecified entries (that is, it is a completed puzzle), `sudoku` will print `correct` or `incorrect` as before.

If the input file does not exist, is not readable, does not follow the format specified above, or contains more than two unspecified elements, `sudoku` may print `error`.

Usage

```
$ ./sudoku file2.txt
solvable
```

1.8 bst: Binary search trees (28 points)

Write a program `bst` that maintains and manipulates a binary search tree according to instructions provided in a file. `bst` takes one argument, which is a path to a file containing the instructions. If the file does not exist, `bst` must print `error` and nothing else.

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 problem has three parts. For part one (9 points), you must implement two operations on the binary search tree: *insert* and *search*. For part two (6 points), you must implement one additional operation: *print*. For part three (9 points), you must implement a second additional operation: *delete*. The remaining 4 points are for style. All three parts are submitted as a single program. You will receive partial credit if your program correctly handles part one or parts one and two.

Input format Each line of the input file contains an instruction. Each line begins with a letter indicating the type of instruction. If the letter is `i` (insert), `s` (search), or `d` (delete), it will be followed by a space and an integer, indicating the argument n. The letter `p` (print) will occur on a line by itself.

If the input file does not exist, is not readable, or does not have the correct format, `bst` may print `error`.

1.8.1 Insertion and searching

For part one, `bst` must implement two operations on binary search trees:

insert n Adds a value n to the tree. The new node will always be added as a child of an existing node (or as the root if the tree is empty). No existing nodes in the tree will move or change values (that is, do not rebalance the tree). If n is already present, `bst` will print `duplicate`. Otherwise, it will print `inserted`. The instruction format is an `i` followed by a space and an 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.

Usage Assume a text file `file1.txt` with the following content:

```
i 10
i 8
s 5
i 5
i 10
s 5
```

With this assumption,

```
$ ./bst file1.txt
inserted
inserted
absent
inserted
duplicate
present
```

1.8.2 Printing

For part two, `bst` must implement a third operation:

print Prints the current tree structure according to the format described below. The instruction format is a `p`.

Output format An empty tree (that is, a `NULL` pointer), is printed as an empty string (that is, nothing). A node is printed as a `(`, followed by the left tree, the value for the node, the right tree, and a `)`.

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

Usage Assume a text file `file2.txt` with the following content:

```
i 10
i 5
s 8
p
i 8
i 10
s 8
p
```

With this assumption,

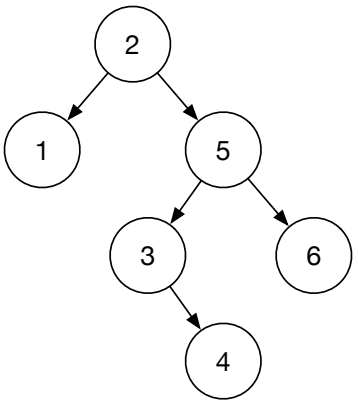
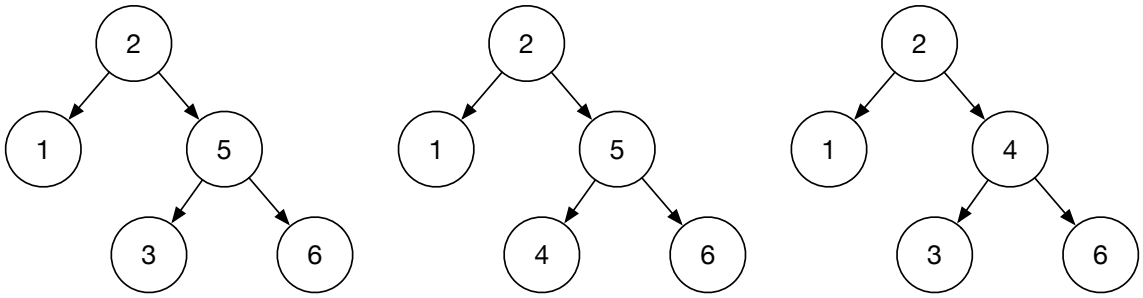


Figure 1: A binary search tree containing six nodes



(a) Deleted 4 (b) Deleted 3 (c) Deleted 5

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

```
$ ./bst file2.txt
inserted
inserted
absent
((5)10)
inserted
duplicate
present
((5(8))10)
```

1.8.3 Deletion

For part three, `bst` must implement a fourth operation:

delete n Remove the value n from the tree. If n is not present, print `absent`. Otherwise, print `deleted`. The instruction format is a `d` followed by a space and an integer n.

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 2(a) shows the result of deleting 4 from the tree in Figure 1.

If a node has one child, it can be replaced by that child. Figure 2(b) shows the result of deleting 3 from the tree in Figure 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 2(c) shows the result of deleting 5 from the tree in Figure 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.

Usage Assume a text file `file3.txt` with the following content:

```
i 10
i 5
s 8
p
i 8
d 6
p
d 5
p
```

With this assumption,

```
$ ./bst file3.txt
inserted
inserted
absent
((5)10)
inserted
absent
((5(8))10)
deleted
((8)10)
```

2 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, and how to use the provided auto-grader.

2.1 Directory structure

Your project should be stored in a directory named `src`, which will contain eight 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 `yell` would be a file `yell.c`, located in the directory `src/yell`.

This diagram shows the layout of a typical project:

```
src
 +- yell
 |   +- Makefile
 |   +- yell.c
 +- rle
```

```
|   +- Makefile
|   +- rle.c
+- anagram
|   +- Makefile
|   +- anagram.c
+- list
|   +- Makefile
|   +- list.c
+- table
|   +- Makefile
|   +- table.c
+- mexp
|   +- Makefile
|   +- mexp.c
+- sudoku
|   +- Makefile
|   +- sudoku.c
+- bst
    +- Makefile
    +- bst.c
```

2.2 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 (invoked when calling `make` with no arguments), should compile the program. An additional target, `clean`, should delete any files created when compiling the program (typically just the compiled program).

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

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

clean:
	rm -f yell
```

Note that the command for compiling `yell` uses GCC warnings and the address sanitizer. You will lose one style point if your makefile does not include these.

Note that the makefile format requires tab characters on the indented lines. Text copied from this document may not be formatted correctly.

2.3 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
```

2.4 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`, and a directory of test cases `data`.

You may create your `src` directory inside `pa1`.

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

```
python grader.py
```

The auto-grader will compile and execute the programs in the directory `src`, if `src` has the structure described in Section 2.1.

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
```

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 `bst`:

```
python grader.py bst
```

You may also 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. This may be useful as a final check before (or just after) submitting your archive.

Note: before testing your program, the auto-grader will execute `make clean` and `make` in the program directory. If either command fails for any reason, such as compiler errors or a missing or invalid makefile, you will receive no points for that program. Before submitting, make sure that you can execute `make clean` and `make` in the program directory on an iLab machine.

3 Grading

This project is worth 100 points, of which 69 will be determined by program performance on test cases and 31 will be determined by examination of source code.

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.

3.1 Style points

Each program may receive up to four style points (or three, in the case of `yell`). These will be awarded by graders as part of a manual inspection of your code. The graders will consider several aspects of your program, such as organization and use of comments.

Some advice for winning points:

- Make sure to use `-Wall -Werror -fsanitize=address` when compiling.
- Use indentation to make your code more readable.
- Document your code with comments. At a minimum, describe the purpose of any functions you create.
- Except for very short programs, do not put all your code in a single function.

Several of the programs can be broken into two layers: an inner layer that manipulates a data structure, and an outer layer that reads instructions from the input and generates output. Being aware of these layers may help you organize your code into functions.

3.2 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.