CS211 Spring 2021
Programming Assignment I

David Menendez

Due: February 25, 2021, 10:00 PM
Hand in by February 26, 2021, 4:00 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 2 describes how your project will be graded, and section 3 describes how to structure and submit your project. In particular, section 3.1 describes how to set up your project and provides an introduction to using the auto-grader. Please read the entire assignment description before beginning the assignment.

Note that the assignment is due at 10:00 PM, but submissions will be accepted without penalty at late as 4:00 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 February 24.

Advice Pay attention to the program descriptions, especially how the programs receive their input. Some programs use only their command-line arguments (`argv`), others read from standard input (with `scanf()` or similar), or from a file. Read the specifications carefully, and think about what is being asked.

Make a plan before you start coding. If you cannot solve the problem yourself, you will not be able to write a program to do it. Don't be afraid to draw diagrams to guide your understanding of the data structures or control flow.

If you have not used command-line shells before, be aware that the text you type at the prompt is processed before being sent to a program as arguments. Most punctuation marks, aside from ., _, and -, have special meanings and must be escaped or enclosed with quotation marks in order to pass them to a program verbatim. You are not responsible for interpreting what the user typed: assume that any strings your program receives are intentional and perform only the processing required by the program specification.

Finally, it is not necessary to print a complete line at a time. If you are printing several strings or other values sequentially, it is both simpler and more efficient to call `printf` multiple times than to allocate a string, write the output to the string, and then print the string all at once. Similarly, it is easier to read input one token at a time rather than reading a line and the breaking the line into tokens.

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

1.1 `whisper`: String operations

Write a program `whisper` which behaves like `echo`, but more quietly. Like `echo`, `whisper` will print its arguments to standard output, separated by spaces, with the following differences

1. If `whisper` receives no arguments, it will print three periods (`...`), representing silence.
2. If `whisper` receives two or more arguments, it will surround its output with parentheses, representing quiet speech.
3. Any upper-case letters in the input will be replaced with lower-case letters.
4. Any exclamation points in the input will be replaced with periods.

Usage

```
$ ./whisper I love NASA
(i love nasa)
$ ./whisper 'I AM YELLING!'
(i am yelling.)
$ ./whisper
...
```

Notes Use `argc` to determine the number of arguments. Print a single space between any two arguments. Arguments may be zero-length strings, which should not be handled specially. Do not attempt to reconstruct what the user may have typed on the shell: work with the strings your program receives.

```
$ ./whisper
...
$ ./whisper ''
()
$ ./whisper a '' b
(a  b)
```

1.2 `anagram`: String operations II

Write a program `anagram` that prints all the letters in its input, sorted alphabetically. This can be useful for finding anagrams, as anagrams will produce identical output.

`anagram` receives a single argument containing a string. Its output will contain the same letters as the argument, converted to lower-case and sorted alphabetically. Non-letter characters, such as whitespace, punctuation, and digits, will be ignored.

Usage

```
$ ./anagram 'genuine class'
aceegilnnssu
$ ./anagram 'Alec Guinness'
aceegilnnssu
$ ./anagram 'The quick brown fox jumps over the lazy dog.'
abcdeeefghhijklmnoooopqrrsttuuvwxyz
```

Notes You are free to use whatever algorithms you find convenient, and do not need to maximize efficiency. If you want to challenge yourself, find an algorithm that will sort the letters in linear time using a fixed number of local variables and no heap allocation.

1.3 `balance`: Strings and stacks

Write a program `balance` that checks whether a string contains correctly nested and balanced parentheses, brackets, and braces. Your program will take a single argument and analyze whether each open delimiter has a corresponding closing delimiter of the correct type.

If the string is balanced, `balance` will print nothing and exit with status `EXIT_SUCCESS`. Otherwise, `balance` will print an error message and exit with status `EXIT_FAILURE`.

Implementation `balance` will maintain a stack of open delimiters. Each time a `(`, `[`, or `{` is encountered in the input, it will push that delimiter onto the stack. Each time a `)`, `]`, or `}` is encountered, `balance` will pop the top delimiter off the stack and check whether it matches the delimiter encountered in the string. If the delimiters do not match, or the stack is empty, `balance` will print the index for the unexpected delimiter and the closing delimiter encountered.

```
$ ./balance ')'
0: )
$ ./balance '([)]'
2: )
```

If the stack is not empty when `balance` reaches the end of the input, it will print the message `open` followed by a list of closing delimiters in the order needed to balance the string.

```
$ ./balance '([{'
open: }])
```

All non-delimiter characters may be ignored.

Notes You are free to use whatever data structures you find convenient. Note that an array can be used to make a stack, if its size is bounded.

The optimal algorithm requires $O(n)$ time and uses $O(n)$ space, where n is the length of the input string.

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.

Note also that the list will not contain duplicate values.

`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.

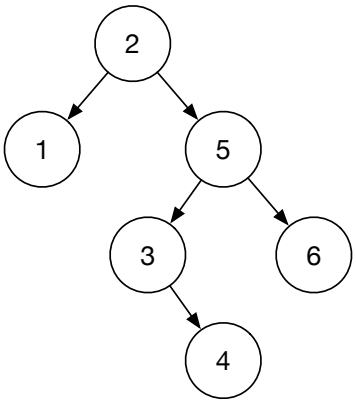


Figure 1: A binary search tree containing six nodes

Usage

```
$ ./bst
i 1
inserted
i 2
inserted
i 1
not inserted
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

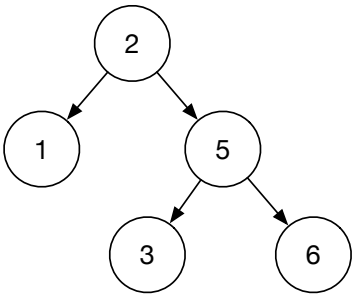
An empty tree (that is, `NULL`) is printed as an empty string. A node is printed as a `(`, followed by the left sub-tree, the item for that node, the right subtree, and `)`, without spaces.

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

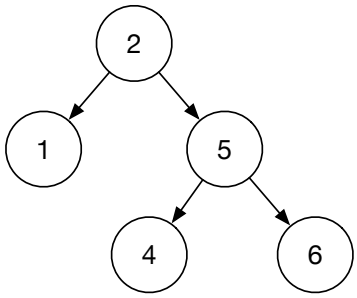
1.6.2 Deleting nodes

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.

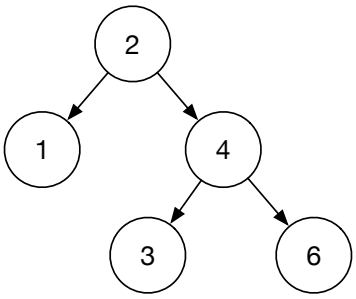
If a node has one child, it can be replaced by that child. Figure 2b shows the result of deleting 3 from the tree in fig. 1. Note that node 4 is now the child of node 5.



(a) Deleted 4



(b) Deleted 3



(c) Deleted 5

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

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.

2 Grading

Your submission will be awarded up to 100 points, based on how many test cases your programs complete successfully.

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 (section 3.2), the requirements for your makefiles (section 3.3), how to create the archive (section 3.4), and how to use the provided auto-grader (section 3.5).

3.1 Getting started

Download the auto-grader and use `tar` to extract it. (The `$` in these examples indicates a prompt. The command you should type comes *after* the prompt and does not include the `$`.)

```
$ tar -xf pa1-grader.tar
```

This will create a directory `pa1/`, containing the auto-grader script and its associated data files.

First, create a subdirectory `pa1/src/`, which will contain a subdirectory for each of the six programs (see section 3.2 for the suggested layout).

```
$ cd pa1
pa1$ mkdir src
```

(Here we are changing the prompt to indicate the working directory. As before, you only type the text after the `$`.)

For each program, you will create a subdirectory with the same name as the program. It will contain a makefile and the source code for your project. For example, to start work on `whisper`, one could create a directory `whisper` inside `pa1/src/` and copy the makefile template into it.

```
pa1$ mkdir src/whisper
pa1$ cp template.make src/whisper/Makefile
```

The template is already set up for `whisper`. For other programs, open the Makefile in the editor of your choice and change the definition `TARGET = whisper` so that `TARGET` is the name of that program. Or, use `sed` to do the copy and edit in a single step.

```
pa1$ sed '{s/whisper/anagram/;}' template.make > src/anagram/Makefile
```

Now create your source code. The template makefile assumes your code will be a single file with the same name as the program. That is, the source for `whisper` will be a file named `whisper.c` in the subdirectory `src/whisper`. You are permitted to modify the makefile to use multiple source files or different source file names, but take care to ensure compatibility with the auto-grader.

Once you have created your source file and are ready to compile, use the auto-grader to create the `build` directory. This is where the auto-grader will place compiled programs and other files. Using a separate build directory keeps your source directory free of clutter.

Running the auto-grader will create the build directory (if it does not exist), compile your program, and run the provided test cases.

```
pa1$ ./grader.py
```

Alternatively, you may use the `--init` option to create the build directories without compiling or running tests.

```
pa1$ ./grader.py --init
pa1$ cd build/whisper
pa1/build/whisper$ make
```

Because the build directory is created from your source code, you are always free to delete it and have the auto-grader reconstruct it.

3.1.1 Testing early and often

The auto-grader provided to you is the same one we will use to test your code. To avoid disaster, make sure that the auto-grader can compile and execute your program! Each time you make progress with a program, run the auto-grader to check for improvement or regression.

By default, the auto-grader will test all the programs. If a list of programs or test groups is provided, the auto-grader will perform only the specified tests. For example, to test only `list`, use:

```
pa1$ ./grader.py list
```

To get more information about failing test cases, use the `--verbose` or `-v` option, which will print the full input to and output from the program on failing test cases. This may be combined with the explicit list of tests to perform.

```
pa1$ ./grader.py -v
pa1$ ./grader.py -v list
```

Use `-v` twice to see input and output for successful test cases.

If the output from `-v` is overwhelming, use `--stop` or `-1` to halt processing after the first failed test case. (Note that `--stop` implies `--verbose`.)

```
pa1$ ./grader.py --stop
pa1$ ./grader.py -1 bst:2
```

3.2 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
 +- whisper
 |   +- Makefile
 |   +- whisper.c
 +- anagram
 |   +- Makefile
 |   +- anagram.c
 +- balance
 |   +- Makefile
 |   +- balance.c
 +- list
 |   +- Makefile
 |   +- list.c
 +- mexp
 |   +- Makefile
 |   +- mexp.c
 +- bst
     +- Makefile
     +- bst.c
```

3.3 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).

The auto-grader script is distributed with an example makefile, which looks like this (note that an actual makefile must use tabs rather than spaces for indentation):

```
TARGET = whisper
CC     = gcc
CFLAGS = -g -std=c99 -Wall -Wvla -Werror -fsanitize=address,undefined

$(TARGET): $(TARGET).c
        $(CC) $(CFLAGS) $^ -o $@

clean:
        rm -rf $(TARGET) *.o *.a *.dylib *.dSYM
```

It is simplest to copy this file into the directories for each program, replacing `whisper` with the name of that specific program. This will ensure that you programs will be compiled with the recommended options.

It is further recommended that you use `make` to compile your programs, rather than invoking the compiler directly. This will ensure that your personal testing is performed with the same compiler settings as the auto-grader. The makefiles created in the build directory by the auto-grader refer to the makefiles you create in the source directory and therefore pick up any changes made.

You may add additional compiler options as you see fit, but you are advised to leave the compiler warnings, sanitizers, and debugger information (`-g`). The makefile shown here specifies the C99 standard, in order to allow C++-style `//` comments; you may change that to C89, if you prefer.

Compiler options The sample makefile uses the following compiler options, listed in the `CFLAGS` make variable:

-g Include debugger information, used by GDB and AddressSanitizer.

-std=c99 Require conformance with the 1999 C Standard. (Disable GCC extensions.)

-Wall Display most common warning messages.

-Wvla Warn when using variable-length arrays.

-Werror Promote all warnings to errors.

-fsanitize=address,undefined Include run-time checks provided by AddressSanitizer and UBSan. This will add code that detects many memory errors and guards against undefined behavior. (Note that these checks discover problems with your code. Disabling them will not make your code correct, even if it seems to execute correctly.)

3.4 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 should be moved or removed.

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

```
pa1$ tar -vzcf 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:

```
pa1$ tar -tf pa1.tar
```

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

```
pa1$ ./grader.py -a pa1.tar
```

3.5 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:

```
pa1$ 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 `pa1`. If you prefer to create `src` outside the `pa1` 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:

```
pa1$ ./grader.py
```

The auto-grader will compile and execute the programs in the directory `src`, assuming `src` has the structure described in section 3.2.

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 `anagram`:

```
pa1$ ./grader.py anagram
```

To stop the auto-grader after the first failed test case, use the `--stop` or `-1` option.

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 `--verbose` or `-v` option:

```
pa1$ ./grader.py -v
```

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

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

```
pa1$ ./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 `--src` or `-s` option. For example,

```
pa1$ ./grader.py -s ../path/to/src
```

Refreshing the build directory In the unlikely event that your build directory has become corrupt or otherwise unusable, you can simply delete it using `rm -r build`. Alternatively, the `--fresh` or `-f` option will delete and recreate the build directory before testing.