CS 211: Computer Architecture, Fall 2019
Programming Assignment 1: Introduction to C

Introduction

This assignment is designed to give you some initial experience with programming in C, as well as compiling, linking, running, and debugging. Your task is to write 6 small C programs. Each of them will test a portion of your knowledge about C programming. They are discussed below. Your program must follow the input-output guidelines listed in each section **exactly**, with no additional or missing output.

First: Array and Sorting (10 Points)

Write a program that will read an array from a file and sort the given array. It should return the array sorted with all even numbers in ascending order at the front followed by all odd numbers in descending order. You may assume that input array will not have more than 20 elements (i.e., max size is 20).

You can use any sorting algorithm you know or wish to use. However, you cannot use the library sort functions. You should write your own sort function.

Input-Output format: Your program will take the file name as input. The first line in the file provides the total number of values in the array. The second line will contain a list of numbers separated by tabs. For example a sample input file "file1.txt" is:

```
6
25 10 1 99 4 2
```

Your output will be the sorted list of numbers, even numbers (ascending) and then odd numbers (descending), each separated by tabs.

```
$ ./first file1.txt
2 4 10 99 25 1
```

We will not give you improperly formatted files. You can assume all your input files will be in proper format as above.

Second: Linked List (20 points)

In this part, you have to implement a linked list that maintains a list of integers in sorted order. Thus, if the list contains 2, 5 and 8, then 1 will be inserted at the start of the list, 3 will be inserted between 2 and 5 and 10 will be inserted at the end. The list can contain duplicate elements.

Input format: This program takes a file name as an argument from the command line. The file is either blank or contains successive lines of input. Each line contains a character, either 'i' or 'd', followed by a tab character and then an integer. For each of the lines that starts with 'i', your program should insert that number in the linked list in sorted order. If it is already there, your program can insert it before or after the existing entry. If the line starts with a 'd', your program should delete the first value if it is present in the linked list. If there are duplicates your program must delete just the first occurrence of the value. Your program should silently ignore the line if the requested value is not present in the linked list.

Output format: At the end of the execution, your program should print the number of nodes in the list in the first line of the output and all unique values of the linked list in sorted order in the next line. Note, while printing the size your program must consider all values, but while printing the value you must print duplicate values just once. The values should be in a single line separated by tabs. There should be no leading or trailing white spaces in the output. Your program should print "error" (and nothing else) if the file does not exist. Your program should print 0 followed by a blank line if the input file is empty or the resulting linked list has no nodes.

Example Execution:

Lets assume we have 3 text files with the following contents:

```
file1.txt is empty

file2.txt:
i 10
i 12
d 10
i 5

file3.txt:
d 7
i 10
i 5
i 10
d 5
```

Then the result will be:

```
$ ./second file1.txt
0
```

```
$ ./second file2.txt
2
5 12
$ ./second file3.txt
2
10
$ ./second file4.txt
error
```

Third: Hash table (20 points)

In this part, you will implement a hash table containing integers. The hash table has 10,000 buckets. An important part of a hash table is collision resolution. In this assignment, we want you to use chaining with a linked list to handle a collision. This means that if there is a collision at a particular bucket then you will maintain a linked list of all values stored at that bucket. For more information about chaining, see `http://research.cs.vt.edu/AVresearch/hashing/openhash.php`.

A hash table can be implemented in many ways in C. You must find a simple way to implement a hash table structure where you have easy access to the buckets through the hash function. As a reminder, a hash table is a structure that has a number of buckets for elements to "hash" into. You will determine where the element falls in the table using the hash function.

You must not do a linear search of the 10,000 element array. We will not award any credit for $O(n)$ time implementation of searches or insertions in the common case.

For this problem, you have to use following hash function: key modulo the number of buckets.

Note that C's modulo operator returns a negative number for negative inputs, so `x % N` will be in the range $[-N+1, N-1]$ instead of $[0, N-1]$. In theory, you could either multiply negative results by -1 or add N to get it into the nonnegative range. For consistency with the autograder, please use the latter method, adding N, so that for example, your hash function should map an input of -5 to bucket 9995.

Input format: This program takes a file name as argument from the command line. The file contains successive lines of input. Each line contains a character, either 'i' or 's', followed by a **tab** and then an integer. For each line that starts with 'i', your program should insert that number in the hash table if it is not present. If the line starts with a 's', your program should search the hash table for that value.

Output format: Your program my print two counts: (1) the number of insertions where collision occurred, and (2) the number of successful searches. In the first line your program should print the number of collisions, i.e. during insertion if the bucket already had some data (may not be the same value) then you need to count that as one collision. In the next line, your program should print the number of searches where the value was present in the hash table. You can assume that the program inputs will have proper structure.

Example Execution:

Lets assume we have a text file with the following contents:

```
file2.txt:
i 10
i 12
s 10
i 10010
s 5
s 10010
```

The the results will be:

```
$ ./third file2.txt
1
2
```

Fourth: Matrix Multiplication (20 Points)

This program will test your ability to manage memory using **malloc()** and provide some experience dealing with 2D arrays in C.

Your task is to create a program that multiplies two matrices and outputs the resulting matrix. The input matrices can be the same or different sizes.

Input-Output format: Your program will take the file name as input. The first line in the file will provide the number of rows and columns in the matrix separated by a tab. The subsequent lines will provide the contents of the matrix. The numbers in the same row are tab separated and the rows are separated with new lines. This will be followed by the same format for the dimensions and content of the second matrix.

For example, a sample input file "file1.txt":

```
2	3
1	2	3
4	5	6
3	2
1	2
3	4
5	6
```

The first number (2) refers to the number of rows and the second number (3) refers to the number of columns in the matrix. The dimensions of the of the first matrix will be 2x3 and the second matrix will be 3x2. The output on executing the program with the above input is shown below. The outputted numbers should be tab separated in the same row with a newline between rows. There should not be extra tabs or spaces at the end of the line or at the end of the file.

```
$ ./fourth file1.txt
```

```
22  28
49  64
```

We will not give you improperly formatted files. You can assume all your input files will be in proper format as above with no matrix having 0 rows or columns.

For matrices that cannot be multiplied your program should output "bad-matrices".

Fifth: String Operations II (10 points)

The fifth part requires you to read an input string representing a sentence, form a word whose letters are all the vowels in the given sentence, and print it. You should preserve the case of the vowels in the input.

Input and output format: This program takes a string of space-separated words, and should output a single word as the output.

```
$ ./fifth Hello World!
eoo
$ ./fifth Welcome to CS211
eoeo
$ ./fifth Rutgers Scarlet Knights
ueaei
```

Sixth: Binary Search Tree (20 points)

In the sixth part, you have to implement a binary search tree. The tree must satisfy the binary search tree property: the key in each node must be greater than all keys stored in the left sub-tree, and smaller than all keys in right sub-tree. You have to dynamically allocate space for each node and free the space for the nodes at the end of the program.

Input format:

This program takes a file name as an argument from the command line. The file is either blank or contains successive lines of input. Each line starts with a character, either 'i', followed by a tab and then an integer. Your program should insert that number in the binary search tree if it is not already there. If it is already present, you will not change the tree.

Output format:

Your program will print all the elements in the tree in ascending order. To print elements in ascending order, you will need to traverse the tree in-order. In an in-order traversal you will visit the left child, then the parent and the right child. You can find more information about in-order traversal at:
`https://en.wikipedia.org/wiki/Tree_traversal#In-order_(LNR)`

The elements have to be printed in a single line separated by tabs. Your program should print "error" (and nothing else) if the input file does not exist.

Example Execution:

Lets assume we have a file file1.txt with the following contents:

```
i   5
i   3
i   4
i   3
i   6
```

Executing the program in the following fashion should produce the output shown below:

```
$ ./sixth file1.txt
3   4   5   6
```

Structure of your submission folder

All files must be included in the `pa1` folder. The `pa1` directory in your tar file must contain 6 subdirectories, one each for each of the parts. The name of the directories should be named first through sixth (in lower case). Each directory should contain a c source file, a header file (if you use it) and a Makefile. For example, the subdirectory first will contain, first.c, first.h (if you create one) and Makefile (the names are case sensitive).

```
pa1
|- first
   |-- first.c
   |-- first.h (if used)
   |-- Makefile
|- second
   |-- second.c
   |-- second.h (if used)
   |-- Makefile
|- third
   |-- third.c
   |-- third.h (if used)
   |-- Makefile
|- fourth
   |-- fourth.c
   |-- fourth.h (if used)
   |-- Makefile
|- fifth
   |-- fifth.c
   |-- fifth.h (if used)
   |-- Makefile
|- sixth
   |-- sixth.c
   |-- sixth.h (if used)
   |-- Makefile
```

A sample Makefile is given below, and you can re-use this for the different programs by changing the name of the output file on the first line:

```
OUTPUT=first
CFLAGS=-g -Wall -Werror -fsanitize=address -std=c99
LFLAGS=-lm

%: %.c %.h
gcc $(CFLAGS) -o $@ $< $(LFLAGS)

%: %.c
gcc $(CFLAGS) -o $@ $< $(LFLAGS)

all: $(OUTPUT)

clean:
rm -f *.o $(OUTPUT)
```

Note that the -fsanitize=address flag is useful for detecting memory errors, but when using valgrind to check for memory leaks, you should omit this option. Otherwise, the two tools conflict with each other and you'll get incorrect results from valgrind (e.g., 32 bytes lost, even if you never call malloc).

Submission

You have to submit the assignment using Canvas. Your submission should be a tar file named `pa1.tar`. To create this file, put everything that you are submitting into a directory (folder) named `pa1`. Then, `cd` into the directory containing `pa1` (that is, `pa1`'s parent directory) and run the following command:

```
tar cvf pa1.tar pa1
```

To check that you have correctly created the tar file, you should copy it (`pa1.tar`) into an empty directory and run the following command:

```
tar xvf pa1.tar
```

This should create a directory named `pa1` in the (previously) empty directory.

The `pa1` directory in your tar file must contain 6 subdirectories, one each for each of the parts. The name of the directories should be named first through sixth (in lower case). Each directory should contain a c source file, a header file (if necessary) and a make file. For example, the subdirectory first will contain, first.c, first.h and Makefile (the names are case sensitive).

AutoGrader

We provide the AutoGrader to test your assignment. AutoGrader is provided as autograder.tar. Executing the following command will create the autograder folder.

```
$ tar xvf autograder.tar
```

There are two modes available for testing your assignment with the AutoGrader.

First mode

Testing when you are writing code with a `pa1` folder

(1) Lets say you have a `pa1` folder with the directory structure as described in the assignment.

(2) Copy the folder to the directory of the autograder

(3) Run the autograder with the following command

```
$ python auto_grader.py
```

It will run your programs and print your scores.

Second mode

This mode is to test your final submission (i.e, pa1.tar)

(1) Copy pa1.tar to the auto_grader directory

(2) Run the auto_grader with pa1.tar as the argument.

The command line is

```
$ python auto_grader.py pa1.tar
```

Scoring

The autograder will print out information about the compilation and the testing process. At the end, if your assignment is completely correct, the score will something similar to what is given below.

```
You scored
10.0  in  second
10.0  in  fourth
10.0  in  third
10.0  in  sixth
5.0  in  fifth
5.0  in  first
Your TOTAL SCORE =  50.0 /50
Your assignment will be graded for another 50 points with test cases not given to you
```

Grading Guidelines

- You should make sure that we can build your program by just running `make`.

- You should test your code as thoroughly as you can. For example, programs should *not* crash with memory errors.
- Your program should produce the output following the example format shown in previous sections. Any variation in the output format can result **in up to 100% penalty**. Be especially careful to not add extra whitespace or newlines. That means you will probably not get any credit if you forgot to comment out some debugging message.

Be careful to follow all instructions. If something doesn't seem right, ask on Piazza.