## 1 Introduction

Basic application programming in C is an essential step down towards the lower levels of computing. This project explores fundamental aspects of getting work done in C:

* Dynamic memory management with malloc()/free()
* Reading data from files in both text and binary format
* Displaying information to the screen
* Reading commands from users in interactive programs
* Dealing with structs and the data structures they can build

The assignment is divided into several problems.

* Problem 1 deals with reading data from files into dynamically allocated arrays
* Problem 2 builds a simple display routine which can be used to display the data read using Problem 1 code
* Problem 3 builds a simple binary search tree application

Each problem is fairly independent from the others so may be worked on in any order though to complete Problem 2, code from Problem 1 must be finished.

### 1.1 Grading Criteria

Credit for this assignment will be given based on two categories.

#### Manual Inspection Criteria (~50%)

Each problem has a checklist of things that graders will look for. The checklist is in the spec and often contains hints on what to do. Make sure you have a look at these.

#### Automated Testing (~50%)

After the release of the project, automated tests will be released for each problem later. Instructions on how to use these will be published in the spec so that you can run the tests yourself to see how many points you are getting. Tests require that code compiles and runs according to the descriptions given so make sure you verify that these work.

### 1.2 Getting Started

Take the following steps to get started

1. Download the code associated with the project linked at the top of the spec. Unzip it and examine some of the provided code.
2. Examine the overview of the files provided listed in the [Download and Setup](http://www-users.cs.umn.edu/~kauffman/2021/a1.html#download) section. This gives brief descriptions of files that already exist and those that you must create.
3. Pick a problem and read. There is a lot of information and many examples provided for each problem. Reading this will help you write the correct code earlier rather than later.
4. Ask questions: if its not clear how to proceed, put up a Piazza post or visit an office hour.
5. Get coding: **don't wait** to start for too long as this will greatly increase your stress levels an potentially result in late submissions.
6. Familiarize yourself with the [late submission policy](http://www-users.cs.umn.edu/~kauffman/2021/syllabus.html#sec-3-5) for assignments so you are not caught off guard. **No submissions will be accepted more than 48 hours after the deadline.**

### 1.3 Makefile

A Makefile is provided as part of this project. Building programs in C is a bit tedious and most folks use **build systems** of which make is the oldest. The instructions and dependencies to create programs are written in a Makefile which is then interpreted by the make program which will run gcc and other commands to create programs.

Use this Makefile as is by issuing commands like make deltas\_main

> make deltas\_main # build problem 1 main program

gcc -Wall -g -c deltas\_main.c

gcc -Wall -g -c read\_deltas.c

gcc -Wall -g -o deltas\_main deltas\_main.o read\_deltas.o

> make clean # remove all programs/binary object files

rm -f save\_deltas deltas\_main print\_graph\_demo graph\_file tree\_main \*.o

> make tree\_main # build problem 3 main program

gcc -Wall -g -c tree\_main.c

gcc -Wall -g -c tree\_funcs.c

gcc -Wall -g -o tree\_main tree\_main.o tree\_funcs.o

> make clean # remove all programs/binary object files

rm -f save\_deltas deltas\_main print\_graph\_demo graph\_file tree\_main \*.o

> make # build all programs/objects for the assignment

gcc -Wall -g -c save\_deltas.c

gcc -Wall -g -o save\_deltas save\_deltas.o deltas.h

gcc -Wall -g -c deltas\_main.c

gcc -Wall -g -c read\_deltas.c

gcc -Wall -g -o deltas\_main deltas\_main.o read\_deltas.o

gcc -Wall -g -c print\_graph\_demo.c

gcc -Wall -g -c print\_graph.c

gcc -Wall -g -o print\_graph\_demo print\_graph\_demo.o print\_graph.o

gcc -Wall -g -c graph\_file.c

gcc -Wall -g -o graph\_file graph\_file.o print\_graph.o read\_deltas.o

gcc -Wall -g -c tree\_main.c

gcc -Wall -g -c tree\_funcs.c

gcc -Wall -g -o tree\_main tree\_main.o tree\_funcs.o

You are not required to understand all that is the Makefile (yet) but it is a very useful tool and may be worth your while to inspect.

### 1.4 Automated Tests

Automated tests can be downloaded in a zip linked at the top of the spec. **These tests are known to work on lab machines only**. They may work on other machines/environments but this is not guaranteed or supported.

Copy the files provided into the project directory. Add the following to the bottom of the Makefile

# Testing Targets

VALGRIND = valgrind --leak-check=full --show-leak-kinds=all

test : test-p1 test-p2 test-p3

test-p1 : test\_read\_deltas

@printf "===TESTS for P1===\n"

@printf "Running binary tests for read\_deltas\n"

./test\_read\_deltas

@printf "\n"

@printf "Running tests in Valgrind\n"

$(VALGRIND) ./test\_read\_deltas

@printf "\n"

test\_read\_deltas.o : test\_read\_deltas.c deltas.h

$(CC) -c $<

test\_read\_deltas : test\_read\_deltas.o read\_deltas.o

$(CC) -o $@ $^

test-p2 : print\_graph\_demo graph\_file save\_deltas

@printf "===TESTS for P2===\n"

@printf "Testing print\_graph via print\_graph\_demo\n"

./test\_print\_graph.sh

@printf "\n"

@printf "Testing graph\_file\n"

./test\_graph\_file.sh

test-p3 : tree\_main

@printf "===TESTS for P3===\n"

./test\_tree\_main.sh

clean-tests :

rm -f test\_read\_deltas

rm -f test-data/\*.tree test-data/\*.valgrind

rm -f test-data/\*.actual test-data/\*.expect test-data/\*.in

rm -f test-data/\*.empty test-data/\*.txt

rm -f test-data/\*.int test-data/\*.diff

This will enable automated tests to be run via calls like

> make test-p1 # run tests for problem 1

gcc -Wall -g -c test\_read\_deltas.c

gcc -Wall -g -c read\_deltas.c

gcc -Wall -g -o test\_read\_deltas test\_read\_deltas.o read\_deltas.o

===TESTS for P1===

Running binary tests for read\_deltas

./test\_read\_deltas

Test 0 text-5 : read\_text\_deltas() len= 5 : OK

Test 1 text-128 : read\_text\_deltas() len= 32 : OK

Test 2 text-one : read\_text\_deltas() len= 1 : OK

...

========================================

10 / 10 tests passed

> make test-p2 # run tests for problem 2

gcc -Wall -g -c print\_graph\_demo.c

gcc -Wall -g -c print\_graph.c

...

===TESTS for P2===

Testing print\_graph via print\_graph\_demo

./test\_print\_graph.sh

Output OK

Valgrind OK

Testing graph\_file

./test\_graph\_file.sh

Loading tests... 10 tests loaded

Running 10 tests

RUNNING NORMAL TESTS

TEST 1 text-ten-5 : OK

TEST 2 text-ten-20 : OK

TEST 3 text-21-5 : OK

TEST 4 text-21-12 : OK

TEST 5 int-21-7 : OK

...

=====================================

OVERALL:

10 / 10 Normal correct

10 / 10 Valgrind correct

> make test # run tests for all problems

...

Each problem describes specifically how tests can be run and how credit will be assigned.

Note that in some cases, testing scripts can run single tests at a time which is useful for debugging the test.

While grading, graders will typically download student code on a lab machine and invoke

make test-pX

where X is a problem number. If the Makefile is missing or incomplete or if code does not compile, **zero credit** will be assigned for the testing grade.

## 2 Download Code and Setup

Download the code pack linked at the top of the page. Unzip this which will create a project folder. Create new files in this folder. Ultimately you will re-zip this folder to submit it.

| **File** | **State** | **Notes** |
| --- | --- | --- |
| Makefile | Provided | Build file to compile all programs |
| save\_deltas.c | Provided | Problem 1 sample to create delta files |
| deltas\_main.c | Provided | Problem 1 main() function |
| deltas.h | Provided | Problem 1 header file |
| read\_deltas.c | Create | Problem 1/2 functions to write |
|  |  |  |
| data/short.txt | Data | Problem 1 data files |
| data/short.int | Data |  |
| data/short.4bit | Data |  |
| data/wave.txt | Data |  |
| data/wave.int | Data |  |
| data/wave.4bit | Data |  |
| print\_graph\_demo.c | Provided | Problem 2 demo program |
| print\_graph.c | Create | Problem 2 function to write |
| graph\_file.c | Create | Problem 2 main function to write |
| tree.h | Provided | Problem 3 header file |
| tree\_funcs.c | Create | Problem 3 functions to write |
| tree\_main.c | Create | Problem 3 main function to write |
|  |  |  |
| data/stranger-demo.script | Data | Problem 3 sample input to main program |
| data/stranger.tree | Data | Problem 3 sample tree saved to file |
| data/big.tree | Data | Problem 3 sample big tree saved to file |
| TESTING |  |  |
| test-data/ | Testing | Directory in which temporary testing files are written |
| test\_read\_deltas.c | Testing | Problem 1 tests for read\_deltas.c |
| test\_print\_graph.sh | Testing | Problem 2 Tests for print\_graph\_demo |
| test\_graph\_file.sh | Testing | Problem 2 Test script for graph\_file |
| test\_graph\_file\_data.sh | Testing | Problem 2 Test data for graph\_file |
| test\_tree\_main.sh | Testing | Problem 3 Test script for tree\_main |
| test\_tree\_main\_data.sh | Testing | Problem 3 Test data for tree\_main |

## 3 Problem 1: Reading Text and File Data

### 3.1 Overview: Delta Arrays

This problem centers around a simple task: read integers from a file into a dynamically allocated array. The caveat to this is that the integers in the file are stored in a special format: an initial value followed by *deltas* or changes from the previous element. An example is below: comments are on the right side

data/short.txt:

20 # start with 20

-2 # subtract 2

-4 # subtract 4

-4 # subtract 4

-3 # subtract 3

-5 # subtract 5

1 # add 1

-2 # subtract 2

4 # add 4

4 # etc...

...

The deltas\_main program will read this file and produce the following output

> ./deltas\_main text data/short.txt

Reading text format

data\_len: 21

# read

0 20

1 18

2 14

3 10

4 7

5 2

6 3

7 1

8 5

9 9

10 5

11 7

12 7

13 5

14 5

15 5

16 7

17 8

18 3

19 7

20 4

Notice the array contents are not exactly what is in the file but rather compute an element by adding on the delta from the file to the previous total, sometimes referred to as a *cumulative sum*.

#### Creating Delta files

The data/ directory contains several delta files. These have extensions indicating the format of the data like .txt for textual and .int for binary int data.

The file save\_deltas.c is provided and allows you to create your own delta files for testing. It is demonstrated below.

> make save\_deltas # builds the save\_deltas program

gcc -Wall -g -c save\_deltas.c

gcc -Wall -g -o save\_deltas save\_deltas.o deltas.h

> ./save\_deltas # show usage of program

usage: ./save\_deltas <format> <filename>

reads input from stdin and outputs in given format to <filename>

if typing numbers in, press Ctrl-D to end input

<format> is one of

text : text ints are in the given filename

int : binary ints are in the given filename

4bit : 4bit binary ints are in the given filename

> ./save\_deltas text deltas.txt # save typed deltas in text format in file deltas.txt

10

12

17

15

10

5

-1

3 # press Ctrl-D to end typed input

wrote 8 ints to deltas.txt in text format

> cat deltas.txt # show contents of file deltas.txt on screen

10 2 5 -2 -5 -5 -6 4

> ./save\_deltas int binary.int # save typed deltas in binary format in file binary.int

1

5

-1

4

wrote 4 ints to binary.int in int format

> cat binary.int # show binary data: it's a mess

^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@

#### Read Delta Files: Code to Write

The file deltas\_main.c is provided and shows how required functions are called to read data. The main work to be done is in read\_deltas.c which will implement several functions for reading data in delta files in different formats.

**All code for this problem** should be written in read\_deltas.c to fill in the definition of the functions described below. As the functions are completed, one can compile the deltas\_main.c program and test it on the provided input files.

### 3.2 Reading Text Integers

In read\_deltas.c, define the following function.

int \*read\_text\_deltas(char \*fname, int \*len);

// Reads integers in text delta format from the file named by fname

// and returns an array of them. The first integer in the file gives

// the starting point and subsequent integers are changes from the

// previous total.

//

// Opens the file with fopen() and scans through its contents using

// fscanf() counting how many text integers exist in it. Then

// allocates an array of appropriate size using malloc(). Uses

// rewind() to go back to the beginning of the file then reads

// integers into the allocated array. Closes the file after reading

// all ints. Returns a pointer to the allocated array.

//

// The argument len is a pointer to an integer which is set to the

// length of the array that is allocated by the function.

//

// If the file cannot be opened with fopen() or if there are no

// integers in the file, sets len to -1 and returns NULL.

Below are some implementation notes explaining some details.

#### Text Delta Format

As described above, the format of files for this function is *text delta* which will have the .txt extension. You can look at these files with a text editor. The first number is the starting point and subsequent numbers are changes from the last number. Some examples:

| delta file: | 0 | 2 | 2 | 2 | 2 | 2 | 1 | 2 | 1 | 2 | 1 |

| array value: | 0 | 2 | 4 | 6 | 8 | 10 | 11 | 13 | 14 | 16 | 17 |

| array index: | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |

| delta file: | 20 | -2 | -4 | -4 | -3 | -5 | 1 | -2 | 4 | 4 | -4 |

| array value: | 20 | 18 | 14 | 10 | 7 | 2 | 3 | 1 | 5 | 9 | 5 |

| array index: | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |

Note that the contents of delta text files can occur on one line, each on a separate line, or several per line. All numbers will be separated by one or more whitespace characters. Make use of the scanning techniques below to handle this without trouble.

#### Opening Text Files

Make use of the standard C file I/O function fopen() to open a named file. Do some research to figure out the arguments to this, particularly

* What arguments are passed to fopen() to open a file for reading
* What the function returns on success
* What the function returns on failure which indicates a file could not be opened.

#### Reading Data from Text Files

Make use of the C function fscanf() which is similar to scanf() to read data. Using a format specifier like %d will skip any whitespace to find the next integer available in the input stream. This allows one to handle data spread across multiple lines easily.

Since the number of integers in the file is not known in advance, use the following **two-pass strategy** to determine the size of the array required.

1. Use a loop with fscanf() just to count how many integers are in the file. Each time you attempt a read, capture the return value of fscanf(). If the return value is EOF (end of file) then the read failed due to reaching the end of the file.
2. Allocate an array of integers big enough to hold the contents of the file in it. Use malloc() for this.
3. Use the standard C rewind() function to move back to the beginning of the file.
4. Use another loop with fscanf() to read through the file again. This time, each read should store an integer in the array.

#### Compiling and Running with deltas\_main.c

When you have completed this function, you may wish to it with deltas\_main.c. You will likely need to take the following steps:

* Comment out the parts of deltas\_main.c which use later functions like read\_int\_deltas() so there are no undefined references.
* Compile the main program. While you can compile the program with gcc manually as in
* > gcc -o deltas\_main deltas\_main.c read\_deltas.c

it is likely that you will want to use the provided Makefile which automatically builds for you as in

> make deltas\_main

gcc -Wall -g -c deltas\_main.c

gcc -Wall -g -c read\_deltas.c

gcc -Wall -g -o deltas\_main deltas\_main.o read\_deltas.o

You may begin examining Makefile as it is an incredibly useful tool for building complex projects.

* Run the program with some data files such as
* > ./deltas\_main text data/short.txt
* Reading text format
* data\_len: 21
* # read
* 0 20
* 1 18
* 2 14
* 3 10
* ...

Note that the program needs to be told the format text as the 2nd argument to run properly.

### 3.3 Reading Binary Integers

In read\_deltas.c, define the following function.

int \*read\_int\_deltas(char \*fname, int \*len);

// Reads integers in binary delta format from the file named by fname

// and returns an array of them. The first integer in the file gives

// the starting point and subsequent integers are changes from the

// previous total.

//

// Integers in the file are in binary format so the size of the file

// in bytes indicates the quantity of integers. Uses the stat() system

// call to determine the file size in bytes which then allows an array

// of appropriate size to be allocated. DOES NOT scan through the file

// to count its size as this is not needed.

//

// Opens the file with fopen() and uses repeated calls to fread() to

// read binary integers into the allocated array. Does delta

// computations as integers are read. Closes the file after reading

// all ints.

//

// The argument len is a pointer to an integer which is set to

// the length of the array that is allocated by the function.

//

// If the file cannot be opened with fopen() or file is smaller than

// the size of 1 int, sets len to -1 and returns NULL.

#### Binary Delta Format

The file format expected by this file is **binary** rather than textual so it is likely not possible to examine the data in text format. The data/ directory contains some example files

* data/short.int : 84 bytes : 21 integers
* data/wave.int : 508 bytes : 127 integers

The data in these are identical to the .txt equivalents:

* The first number is the starting point
* Subsequent numbers are changes of the previous number

However, rather than use ASCII characters that are human readable, each integer is stored directly as 4 bytes in the file.

#### Determining File Size / Array Elements

Since each integer occupies 4 bytes directly in the file, the size of the file corresponds to the number of integers in the file. This allows one to allocate an appropriately sized array based on the file size. This means that no scanning is required to determine how many ints are in the file.

The stat() system call in Unix provides information about a file by filling in values of a struct. The file size in bytes is one such piece of information. Use a calling sequence like the following to determine the binary file size.

struct stat sb; // struct to hold

int result = stat(fname, &sb); // unix system call to determine size of named file

if(result==-1 || sb.st\_size < sizeof(int)){ // if something went wrong or bail if file is too small

return NULL;

}

int total\_bytes = sb.st\_size; // size of file in bytes

Note the use of the struct stat sb, on old convention for declaring a struct as a stack variable. Its memory address is passed to stat() which fills in information. The field st\_size is the number of bytes in the file. This can then be used to determine how many ints are in the file (4 bytes per int) and allocate an array with malloc()prior to reading through the file.

Note that this technique only works with binary files. Text files use a variable number of characters/bytes per integer and include formatting characters such as spaces and newlines. This means the number of bytes in the file does not correspond to the number of integers in it.

#### Reading Binary Data

Files with binary data can be opened identically to the text data but reading requires use of the fread() function. This function differs greatly from fscanf() as it has no format string. Instead, it is meant to read raw bits/bytes from a file into a specified memory address. It is likely that you will use a call similar to the following.

fread(&values[i], sizeof(int), 1, fin);

Address to | bytes for |quantity |file handle

store data | one element |to read |to read from

This call reads a single binary integer into the an array element from an open file handle.

Make sure as you read integers into the array you perform the delta computation of adding on the previous value to the recently read delta.

### 3.4 OPTIONAL: Reading 4-bit Signed Integers

For those looking for a bit more challenge, optionally implement the following function (pride only: no bonus credit).

int \*read\_4bit\_deltas(char \*fname, int \*len);

// Reads integers in 4bit delta format from the file named by fname

// and returns an array of them. The first integer in the file gives

// the starting point and subsequent integers are changes from the

// previous total.

//

// Integers in the file are in 4bit format so the size of the file in

// bytes indicates the quantity of integers: 1 4byte integer appears

// at the beginning and subsequently, each byte as two 4bit signed

// integers packed into it. Uses the stat() system call to determine

// the file size in bytes which then allows an array of appropriate

// size to be allocated. DOES NOT scan through the file to count its

// size as this is not needed.

//

// Opens the file with fopen() and uses repeated calls to fread() to

// read binary integers into the allocated array. Does delta

// computations as integers are read. Closes the file after reading

// all ints.

//

// The argument len is a pointer to an integer which is set to

// the length of the array that is allocated by the function.

//

// If the file cannot be opened with fopen() or file is smaller than

// the size of 1 int, sets len to -1 and returns NULL.

#### 4-bit Delta Files

In many applications such as measuring temperature changes every minute, one does not expect large changes. This means a full 32-bit signed integer to represent the change is a waste of space. Instead, a 4-bit signed integer is sufficient to represent deltas between -8 and +7. The 4-bit delta format is as follows.

* The first 4 bytes are a 32 bit integer which is the starting value. This is identical to the previous file formats
* Subsequently, each byte (8 bits) contains TWO 4 bit signed integers which are changes from the previous value.

Like the binary int format above, this format has a direct correspondence of the size of the file to the number of integers

* data/short.int : 21 integers = 4 byte int + (20 ints)/2 ints per byte = 14 bytes.
* data/wave.int : 127 integers = 4 byte int + (126 ints)/2 ints per byte = 67 bytes

This is a significantly compressed size over the sizes of other formats which is useful if space is tight (ex: a remotely located temperatures sensor on top of Mt. Everest).

The trade-off for the space savings is that the 4-bit format requires significantly more skill to decode into a standard array of integer values.

#### Decoding 4-bit Integers

Note that reading 4-bit ints can be done with fread() as it binary was read in the previous problem. However, one will need to read 1 byte at time and process it to become two integer deltas.

As an example, consider the a file starting in the following way:

Byte 1 Byte 2 Byte 3 Byte 4

0000 0111 0000 0000 0000 0000 0000 0000 // little endian 32 bit int = 7

Byte 5

0011 0101 // delta1: right 4 bits = +3, delta2: left 4 bits = +5

Byte 6

1111 0001 // delta1: right 4 bits = -3, delta2: left 4 bits = +1

Byte 7

0111 1000 // delta1: right 4 bits = -8, delta2: left 4 bits = +7

To handle this file format, one would take the following approach.

* Read the initial value as normal (4 bytes for an int)
* Read 1 byte
* Mask the low 4 bits for the first 4-bit delta
* Shift and mask upper 4 bits for the second 4-bit delta
* Shift both deltas left then right to carry the sign bit (0 positive, 1 negative) across the number.

The following C binary operators are useful for this

int result = 0;

// initializ result: 32 bits all 0

result = x & 0b1111;

// result is bitwise and of x and the binary constant of zeroes with 4

// low-order ones

result = x >> 4;

// result is x shifted 4 bits to the right

result = (x << 28) >> 28;

// result is x shifted left 28, the right 28 to carry its sign bit

// across 32 bits

Combining these bitwise operations will allow one to convert a 4-bit signed integer to a 32-bit signed integer.

### 3.5 Manual Inspection Criteria for read\_deltas.c   GRADING

The following criteria will be checked in read\_deltas.c

| **Weight** | **Criteria** |
| --- | --- |
| 10 | read\_text\_deltas() |
|  | Clear loop to count number of elements in file prior to allocation |
|  | Correctly reading data and checking for end of file with fscanf() |
|  | Proper use of malloc() to allocate an appropriately sized array |
|  | Proper use of dereference to set the len parameter |
|  | File closed prior to return |
|  | Clear efforts to check for file open failures or no integers to return NULL |
| 10 | read\_int\_deltas() |
|  | Use of stat() system call to calculate file size |
|  | Use of malloc() to allocate correctly sized array |
|  | Clear loop using fread() to read binary data into the array |
|  | Proper use of dereference to set the len parameter |
|  | File closed prior to return |
|  | Clear efforts to check for file open failures or no integers to return NULL |

### 3.6 Tests for read\_deltas.c   GRADING

| **Weight** | **Criteria** |
| --- | --- |
| 10 | test\_read\_deltas.c |
|  | Provides 10 tests for read\_text\_deltas() and read\_int\_deltas() |
|  | Compile and run using make test-p1 |
|  | 1 point per test passed |
| 5 | Valgrind with test\_read\_deltas.c |
|  | Compile and run using make test-p1 |
|  | 5 points awarded for Valgrind identifying no memory errors |
|  | Deductions made based on severity of memory errors identified by Valgrind |

## 4 Problem 2: Graphing Data

### 4.1 Overview

Prior to advent of graphical displays, computers were mostly operated via text terminals. This meant any graphical displays had to be rendered with character data in a grid. Anyone thinking this format limited the creativity of computer folks is encouraged to run

telnet towel.blinkenlights.nl

which will likely change their tune.

The purpose of this problem is somewhat more mundane: create a simple plotting routine for the text terminal. This print\_graph() function which resides in the print\_graph.c file will display an array of numbers on the screen in a graph-like fashion as per the examples below.

> make print\_graph\_demo # build the provided demo program

gcc -Wall -g -c print\_graph\_demo.c

gcc -Wall -g -c print\_graph.c

gcc -Wall -g -o print\_graph\_demo print\_graph\_demo.o print\_graph.o

> ./print\_graph\_demo # run the provided demo program

DEMO DATA 1

===========

length: 22

min: 0

max: 20

range: 20

max\_height: 10

units\_per\_height: 2.00

+----+----+----+----+-

20 | X

18 | XX

16 | XXX

14 | XXXX

12 | XXXXX

10 | X XXXXXXX

8 | XXX XXXXXXXX

6 | XXXXX XXXXXXXXX

4 | XXXXXXX XXXXXXXXXX

2 | XXXXXXXXXXXXXXXXXXXX

0 |XXXXXXXXXXXXXXXXXXXXXX

+----+----+----+----+-

0 5 10 15 20

DEMO DATA 2

===========

length: 22

min: 0

max: 20

range: 20

max\_height: 20

units\_per\_height: 1.00

+----+----+----+----+-

20 | X

19 | X

18 | XX

17 | XX

16 | XXX

15 | XXX

14 | XXXX

13 | XXXX

12 | XXXXX

11 | XXXXX

10 | X XXXXXXX

9 | XX XXXXXXX

8 | XXX XXXXXXXX

7 | XXXX XXXXXXXX

6 | XXXXX XXXXXXXXX

5 | XXXXXX XXXXXXXXX

4 | XXXXXXX XXXXXXXXXX

3 | XXXXXXXX XXXXXXXXXX

2 | XXXXXXXXXXXXXXXXXXXX

1 | XXXXXXXXXXXXXXXXXXXXX

0 |XXXXXXXXXXXXXXXXXXXXXX

+----+----+----+----+-

0 5 10 15 20

DEMO DATA 3

===========

length: 50

min: 13

max: 996

range: 983

max\_height: 10

units\_per\_height: 98.30

+----+----+----+----+----+----+----+----+----+----

996 | X

897 | X X X X

799 | X X X X X X X X X

701 | XX X X X XXX X XX XXX X X XX

602 | XX X X XX XXX X X XX XXXX XX X XX

504 | XX XXX XX XXX XX X XXX XXXX XX XX X XX

406 | XX X XXX XXXX XXX XX X XXX XXX XXXXXXXXXX X XX

307 | XXX X XXX XXXXXXXXXXX X XXXX XXXXXXXXXXXXXXX X XX

209 | XXX XXXXXXXXXXXXXXXXX XXXXXX XXXXXXXXXXXXXXXXX XX

111 | XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

13 |XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

+----+----+----+----+----+----+----+----+----+----

0 5 10 15 20 25 30 35 40 45

DEMO DATA 4

===========

length: 50

min: 13

max: 996

range: 983

max\_height: 18

units\_per\_height: 54.61

+----+----+----+----+----+----+----+----+----+----

996 | X

941 | X X

886 | X X X X

832 | X X X X X X

777 | X X X X XXX X X XXX XX

722 | XX X X X XXX X X XXX XX

668 | XX X X X XXX X XX XXXX X X XX

613 | XX X X XX XXX X XX XXXX XX X XX

559 | XX XXX XX XXX XX X XX XXXX X XX X XX

504 | XX XXX XX XXX XX X XXX XXXX XX XX X XX

449 | XX X XXX XXXX XXX XX X X XXX XXXX XX XX X XX

395 | XXX X XXX XXXX XXX XX X XXX XXX XXXXXXXXXX X XX

340 | XXX X XXX XXXXXXXXXXX X XXX XXXXXXXXXXXXXXX X XX

286 | XXX X XXX XXXXXXXXXXX X XXXX XXXXXXXXXXXXXXX X XX

231 | XXX X XXXXXXXXXXXXXXX XXXXXX XXXXXXXXXXXXXXXXX XX

176 | XXX XXXXXXXXXXXXXXXXX XXXXXX XXXXXXXXXXXXXXXXXXXX

122 | XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

67 | XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

13 |XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

+----+----+----+----+----+----+----+----+----+----

0 5 10 15 20 25 30 35 40 45

### 4.2 A Function For Graphing Data

In the file print\_graph.c, write a the following C function.

void print\_graph(int \*data, int len, int max\_height);

// Prints a graph of the values in data which is an array of integers

// that has len elements. The max\_height argument is the height in

// character rows that the maximum number data[] should be. A sample

// graph is as follows:

//

// length: 50

// min: 13

// max: 996

// range: 983

// max\_height: 10

// units\_per\_height: 98.30

// +----+----+----+----+----+----+----+----+----+----

// 996 | X

// 897 | X X X X

// 799 | X X X X X X X X X

// 701 | XX X X X XXX X XX XXX X X XX

// 602 | XX X X XX XXX X X XX XXXX XX X XX

// 504 | XX XXX XX XXX XX X XXX XXXX XX XX X XX

// 406 | XX X XXX XXXX XXX XX X XXX XXX XXXXXXXXXX X XX

// 307 | XXX X XXX XXXXXXXXXXX X XXXX XXXXXXXXXXXXXXX X XX

// 209 | XXX XXXXXXXXXXXXXXXXX XXXXXX XXXXXXXXXXXXXXXXX XX

// 111 | XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

// 13 |XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

// +----+----+----+----+----+----+----+----+----+----

// 0 5 10 15 20 25 30 35 40 45

#### Computing Range Statistics

The first step to creating the graph is to compute the minimum and maximum values in the data array. This allows one to compute the range of the data. The max\_height argument can be then be used to compute how many "units" a data value must have to get an X character at the given heights. All this information is printed prior to the main body of the graph as in the following.

length: 50

min: 13

max: 996

range: 983

max\_height: 10

units\_per\_height: 98.30

^^

2 decimal places

Note that the units\_per\_height value is a floating point value and it is printed with 2 decimal places of accuracy.

#### Top and Bottom Axes

The top and bottom axes are simply dashed - lines with a + every 5 steps similar to the following:

+----+----+----+----+----+----+----+----+----+----

The left part of the axis is padded with 5 spaces to allow room for vertical axis numbering.

Below the bottom axis, print an index reference every 5 units. This allows particular data elements (data[11] for example) to be located easily. This is easiest to do the a format specifier like %-5d which will left justify the integers in a field of 5 characters. The bottom axis and numbering should look like

+----+----+----+----+----+----+----+----+----+----

0 5 10 15 20 25 30 35 40 45

^^^^^ ^^^^^

5 spaces 5 chars wide, left justified

#### Main plot Body

The main plot body can be drawn according to the following algorithm.

1. Loop from max\_height down to 0. Each iteration will draw a "level" of the plot
2. At the start of each loop, compute the units at the iteration's level using arithmetic like
3. min + level \* units\_per\_height

Make sure that this is an integer.

1. Print the vertical axis reference number first. Use a format specifier like %3d | which will print the level number reference right justified in a field of 3 characters followed by a space and a vertical bar.
2. Loop through each element of the data array. If the data element is larger than or equal to the unit level, print an X. Otherwise, print a space.
3. After printing X/space for each data, print a newline to move down one level.

Your code should use a doubly nested loop to get the printing correct.

#### Demoing with print\_graph\_demo.c

A correct implementation of print\_graph() should compile against print\_graph\_demo.c and produce the output shown at the top of this section.

### 4.3 Main Function for Graphing

The print\_graph() function should work seamlessly to print the data array produced by the functions in read\_deltas.c. In the file graph\_file.c, create a main()function which does this.

* Accepts 3 command line arguments
  1. A format of a file
  2. A filename
  3. A number which is the maximum height of the graph in characters
* Uses the atoi(str) function to convert the maximum height command line argument from a string to an integer.
* Uses an appropriate function from read\_deltas.c to read data from the given file.
* Plots the data using print\_graph()
* Frees the data array prior to exiting.

Note that this main() function is VERY similar to the main() in deltas\_main.c which means you can lift a lot of code from there to get the job done.

A demonstration of how this program should work is below. follows.

You DO NOT need implement the 4bit file format if you did not do the optional problem on this.

> make graph\_file # build using Makefile

gcc -Wall -g -c graph\_file.c

gcc -Wall -g -c print\_graph.c

gcc -Wall -g -c read\_deltas.c

gcc -Wall -g -o graph\_file graph\_file.o print\_graph.o read\_deltas.o

> ./graph\_file # no arguments gives usage message

usage: graph\_file <format> <filename> <height>

<format> is one of

text : text ints are in the given filename

int : binary ints are in the given filename

4bit : 4bit binary ints are in the given filename

> ./graph\_file text data/short.txt 5 # graph the text short.txt file, height 5

Reading text format

length: 21

min: 1

max: 20

range: 19

max\_height: 5

units\_per\_height: 3.80

+----+----+----+----+

20 |X

16 |XX

12 |XXX

8 |XXXX X X

4 |XXXXX XXXXXXXXXX XX

1 |XXXXXXXXXXXXXXXXXXXXX

+----+----+----+----+

0 5 10 15 20

> ./graph\_file text data/short.txt 20 # graph the text short.txt file, height 20

Reading text format

length: 21

min: 1

max: 20

range: 19

max\_height: 20

units\_per\_height: 0.95

+----+----+----+----+

20 |X

19 |X

18 |XX

17 |XX

16 |XX

15 |XX

14 |XXX

13 |XXX

12 |XXX

11 |XXX

10 |XXXX

9 |XXXX X

8 |XXXX X X

7 |XXXXX X XX XX X

6 |XXXXX X XX XX X

5 |XXXXX XXXXXXXXXX X

4 |XXXXX XXXXXXXXXX XX

3 |XXXXX X XXXXXXXXXXXXX

2 |XXXXXXX XXXXXXXXXXXXX

1 |XXXXXXXXXXXXXXXXXXXXX

1 |XXXXXXXXXXXXXXXXXXXXX

+----+----+----+----+

0 5 10 15 20

> ./graph\_file int data/short.int 10 # graph the binary short.int file, height 10

Reading binary int format

length: 21

min: 1

max: 20

range: 19

max\_height: 10

units\_per\_height: 1.90

+----+----+----+----+

20 |X

18 |XX

16 |XX

14 |XXX

12 |XXX

10 |XXXX

8 |XXXX X X

6 |XXXXX X XX XX X

4 |XXXXX XXXXXXXXXX XX

2 |XXXXXXX XXXXXXXXXXXXX

1 |XXXXXXXXXXXXXXXXXXXXX

+----+----+----+----+

0 5 10 15 20

> ./graph\_file int data/wave.int 15 # graph the binary wave.int file, heigh 15

Reading binary int format

length: 127

min: -20

max: 20

range: 40

max\_height: 15

units\_per\_height: 2.67

+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+-

20 | XXXX XXXX

17 | XXXXXXXXXXXX XXXXXXXXXXXX

14 | XXXXXXXXXXXXXXXXX XXXXXXXXXXXXXXXXX

12 | XXXXXXXXXXXXXXXXXXX XXXXXXXXXXXXXXXXXXX

9 | XXXXXXXXXXXXXXXXXXXXXXX XXXXXXXXXXXXXXXXXXXXXXX

6 | XXXXXXXXXXXXXXXXXXXXXXXXXX XXXXXXXXXXXXXXXXXXXXXXXXXX

4 | XXXXXXXXXXXXXXXXXXXXXXXXXXXX XXXXXXXXXXXXXXXXXXXXXXXXXXXX

1 | XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

-1 |XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX X

-4 |XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX XXX

-6 |XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX XXXX

-9 |XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX XXXXXX

-12 |XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX XXXXXXX

-14 |XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX XXXXXXXXX

-17 |XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX XXXXXXXXXXX

-20 |XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+-

0 5 10 15 20 25 30 35 40 45 50 55 60 65 70 75 80 85 90 95 100 105 110 115 120 125

### 4.4 Manual Inspection Criteria for print\_graph.c and graph\_file.c   GRADING

The following criteria will be checked in read\_deltas.c

| **Weight** | **Criteria** |
| --- | --- |
| 5 | print\_graph.c |
|  | Clear commenting indicating what part of the graph is being printed througout |
|  | Block which prints the initial statistics such as min/max/range |
|  | Clearly marked blocks for printing top and bottom axis |
|  | Clear doubly-nested loop structure to print main graph body |
| 5 | graph\_file.c |
|  | Clear set of cases which selects the appropriate file format/read function |
|  | Both text and binary int formats are handled |
|  | Memory allocated by the read function is free()'d in main() |

### 4.5 Tests for print\_graph.c and graph\_file.c   GRADING

| **Weight** | **Criteria** |
| --- | --- |
| 5 | test\_print\_graph.sh |
|  | Script which checks output of print\_graph\_demo |
|  | Compile and run using make test-p2 OR |
|  | Run individually by doing ./test\_print\_graph.sh |
|  | 0-4 points assigned based on matching expected output |
|  | 1 point assigned for avoiding memory problems identified by Valgrind |
| 10 | test\_graph\_file.sh |
|  | 10 tests for graph\_file executable |
|  | Compile and run using make test-p2 OR |
|  | Run script using ./test\_graph\_file.sh |
|  | 1 point per test passed |
|  | 0-3 point deduction if Valgrind identifies memory problems |
|  | NOTE: individual tests can be run by doing |
|  | ./test\_graph\_file.sh 4 # runs test 4 |

The test\_graph\_file.sh script creates outputs in the test-data/ directory such as test-data/actual.txt which is the output on test which may be useful to inspect. Running a single test as in

./test\_graph\_file.sh 4

ensures that files in the test-data/ directory correspond to the results of that single test.

## 5 Problem 3: Binary Search Trees In C

### 5.1 Overview

This problem implements a rudimentary binary search tree in C along with a program that uses it. The architecture is similar to a lab problem involving linked lists so reviewing that code can provide some insight. The intent is develop a small application which behaves as the following demo indicates.

> make tree\_main # build with Makefile

gcc -Wall -g -c tree\_funcs.c

gcc -Wall -g -o tree\_main tree\_main.o tree\_funcs.o

> ./tree\_main # run main program

BST> Demo

Commands:

print: shows contents of the tree in sorted order

show: shows contents in pre-order indicating its structure

clear: eliminates all elements from the tree

quit: exit the program

add <name>: inserts the given string into the tree, duplicates ignored

find <name>: prints FOUND if the name is in the tree, NOT FOUND otherwise

save <file>: writes the contents of the tree in pre-order to the given file

load <file>: clears the current tree and loads the one in the given file

BST> add Lucas # add some data

BST> add Mike

BST> add Dustin

BST> add Will

BST> add El

BST> print # print the tree in sorted order

Dustin

El

Lucas

Mike

Will

BST> show # show the tree in pre-order with indents

Lucas # root

Dustin # left of root

El # left-right of root

Mike # right of root

Will # right-right of root

BST> find Mike

FOUND

BST> find Nancy

NOT FOUND

BST> add Nancy

BST> show # show tree in pre-order

Lucas # root

Dustin # left of root

El # left-right of root

Mike # right of root

Will # right-right of root

Nancy # right-right-left of root

BST> find Max # look up Max

NOT FOUND

BST> add Max # add Max

BST> show # show tree in pre-order

Lucas

Dustin

El

Mike

Max

Will

Nancy

BST> print # print tree in sorted order

Dustin

El

Lucas

Max

Mike

Nancy

Will

BST> find Barb # look up Barb

NOT FOUND

BST> add Barb # add Barb

BST> show # show tree in pre-order

Lucas

Dustin

Barb

El

Mike

Max

Will

Nancy

BST> save stranger.tree # save tree in file "stranger.tree"

BST> clear # clear all nodes from tree

BST> print # print shows nothing: emtpy tree

BST> add Demigorgon # add element

BST> show # show tree: 1 element

Demigorgon

BST> load stranger.tree # load tree from file which clears current tree

BST> show # restores previous tree

Lucas

Dustin

Barb

El

Mike

Max

Will

Nancy

BST> add Jim # add Jim

BST> add Joyce # add Joyce

BST> show # show tree in pre-order

Lucas

Dustin

Barb

El

Jim

Joyce

Mike

Max

Will

Nancy

BST> load stranger.tree # load tree which clears current tree

BST> show # recently added Jim/Joyce no longer there

Lucas

Dustin

Barb

El

Mike

Max

Will

Nancy

BST> quit # exit program

### 5.2 tree\_funcs.c: tree functions

Examine the header file tree.h to see the two C structs which will be used to represent trees.

// Type of tree nodes

typedef struct node {

char name[128]; // node data

struct node \*left; // left branch, NULL if not present

struct node \*right; // right branch, NULL if not present

} node\_t;

// Type of tree itself

typedef struct {

node\_t \*root; // root of tree, NULL if tree empty

int size; // number of nodes in tree

} bst\_t;

Standard operations are supported by the tree.

* Adding elements
* Searching for an element
* Clearing the entire tree
* Printing tree in-order (sorted) and pre-order (root first, then children)
* Saving and loading from a file

These are broken down into the following C functions which you will need to implement in tree\_funcs.c. Keep in mind that to honor the binary search tree property (left less than / right greater than), you will need to use the return values of strcmp() function calls as string data is present in the tree.

void bst\_init(bst\_t \*tree);

// Initialize the tree to have a null root nodes and have size 0.

int bst\_insert(bst\_t \*tree, char name[]);

// Inserts name into the specified BST. If the name already exists in

// the tree, returns 0 and does not change the BST. Otherwise inserts

// the name in a new node, increments the size of the tree, and

// returns 1. Makes use of the recursive node\_insert function.

node\_t \*node\_insert(node\_t \*cur, char name[]);

// Recursive helper function to insert the given name in the tree

// rooted at cur. If cur is NULL, allocates a new node, copies name

// into it, and returns the new node. If cur is not NULL, checks for

// differences between name and cur.name. For identical data, returns

// NULL. Otherwise recurses left or right to link a new node. Returns

// the node cur to stitch the tree back together as the recursion

// unwinds.

int bst\_find(bst\_t \*tree, char name[]);

// Determines whether name is present or not in the tree using binary

// search. Returns 1 if name is present and 0 otherwise. Uses helper

// node\_find().

int node\_find(node\_t \*cur, char name[]);

// Recursive helper function that uses binary search on tree rooted at

// node. Returns 1 if the name is found somewhere in this tree and 0

// otherwise. Checks the current node for name. If not found, chooses

// left or right subtrees to search according to the sorting order of

// the name compared to the cur node.

void bst\_clear(bst\_t \*tree);

// Eliminate all nodes in the tree setting its contents empty. Uses

// recursive node\_remove\_all() function to free memory for all nodes.

void node\_remove\_all(node\_t \*cur);

// Recursive helper function which visits all nodes in a tree and

// frees the memory associated with them. This requires a post-order

// traversal: visit left tree, visit right tree, then free the cur

// node.

void bst\_print\_inorder(bst\_t \*tree);

// Prints the elements of the tree in sorted order all at the same

// indentation line, one element per line. Makes use of

// node\_print\_inorder().

void node\_print\_inorder(node\_t \*cur);

// Recursive helper function which prints all elements in the tree

// rooted at cur in order: traverses left subtree, prints cur node,

// traverses right tree.

void bst\_print\_preorder(bst\_t \*tree);

// Print all the data in the tree in pre-order with indentation

// corresponding to the depth of the tree. Makes use of

// node\_write\_preorder() for this.

void bst\_save(bst\_t \*tree, char \*fname);

// Saves the tree by opening the named file, writing the tree to it in

// pre-order with node\_write\_preorder(), then closing the file.

void node\_write\_preorder(node\_t \*cur, FILE \*out, int depth);

// Recursive helper function which writes/prints the tree in pre-order

// to the given open file handle. The parameter depth gives how many

// spaces to print before the current node data. Depth increases by 1

// on each recursive call. The function prints the cur node data,

// traverses the left tree, then traverses the right tree.

int bst\_load(bst\_t \*tree, char \*fname );

// Clears the given tree then loads new elements to it from the

// named. Repeated calls to bst\_insert() are used to add strings read

// from the file. If the tree is stored in pre-order in the file, its

// exact structure will be restored.

### 5.3 tree\_main.c: main function / application

In tree\_main.c implement an interactive program which allows users to type commands to manipulate a binary tree.

The provided Makefile should compile the tree program as follows.

> make tree\_main # Compile with Makefile

gcc -Wall -g -c tree\_main.c

gcc -Wall -g -c tree\_funcs.c

gcc -Wall -g -o tree\_main tree\_main.o tree\_funcs.o

> ./tree\_main # Run tree\_main

BST Demo

Commands:

print: shows contents of the tree in sorted order

show: shows contents in pre-order indicating its structure

clear: eliminates all elements from the tree

quit: exit the program

add <name>: inserts the given string into the tree, duplicates ignored

find <name>: prints FOUND if the name is in the tree, NOT FOUND otherwise

save <file>: writes the contents of the tree in post order to the given file

load <file>: clears the current tree and loads the one in the given file

BST>

The following sections provide some implementation details.

#### Read commands, Execute

The basic flow of tree\_main.c follows a provided lab exercise code closely.

* Create a tree variable, likely on the stack as a local variable in main()
* Start a loop that terminates when the user exits or there is no more input
* Each time the user enters a string, read it and check for one of the built-in commands
* On identifying the command, potentially read another string if needed (commands like add and find)
* Call an appropriate bst\_XXX() function to handle the command

#### Supported Commands

To indicate to users of the program the supported commands, use the following code to print out the initial option list.

printf("BST Demo\n");

printf("Commands:\n");

printf(" print: shows contents of the tree in sorted order\n");

printf(" show: shows contents in pre-order indicating its structure\n");

printf(" clear: eliminates all elements from the tree\n");

printf(" quit: exit the program\n");

printf(" add <name>: inserts the given string into the tree, duplicates ignored\n");

printf(" find <name>: prints FOUND if the name is in the tree, NOT FOUND otherwise\n");

printf(" save <file>: writes the contents of the tree in post order to the given file\n");

printf(" load <file>: clears the current tree and loads the one in the given file\n");

**NOTE:** The help message for save incorrectly mentions a post-order traversal but use the pre-order traversal encoded in bst\_save() and node\_write\_preorder().

#### Paths to Tree files for Saving and Loading

Saving and loading trees involves writing to files. The names associated with the files must be specified as a *path* so that if tree files are to be saved or loaded from subdirectories, include this as part of the path. For example, the stranger.tree file is provided in the data/ directory so can be loaded as follows.

> ./tree\_main

...

BST> load data/stranger.tree # load the provided tree from a subdirectory

BST> show # show the tree

Lucas

Dustin

Barb

El

Mike

Max

Will

Nancy

BST> add Demigorgon # add more data

BST> save cur.tree # save in the current directory

BST> clear

BST> show

BST> load cur.tree # load cur.tree from the current directory

BST> show

Lucas

Dustin

Barb

Demigorgon # added prior to the save

El

Mike

Max

Will

Nancy

BST> add Jim # add more data

BST> save data/other.tree # save to a new file in data/ subdir

BST> quit

#### Echoing Commands: -echo option to tree\_main

Some users may wish to "script" this the tree program. An example of such a script is in data/stranger-demo.script which looks like a lot of user commands:

add Lucas

add Mike

add Dustin

add Will

add El

print

show

find Mike

find Nancy

add Nancy

show

find Max

add Max

show

print

find Barb

add Barb

show

save stranger.tree

clear

print

add Demigorgon

show

load stranger.tree

show

add Jim

add Joyce

show

load stranger.tree

show

quit

This can be fed directly to the program without needing type it using Unix pipes as per the following:

> cat data/tree-demo.script | ./tree\_main -echo

Notice the use of a command line argument for tree\_main: the -echo option. This is a REQUIRED feature which prints commands typed by users to the screen. To implement this, do the following.

Prior to main(), insert the following to bits which will make echoing convenient.

// global variable to control echoing, 0: echo off, 1: echo on

int ECHO = 0;

// Print the given string if echoing is turned on

void echo(char \*s){

if(ECHO){

printf(s);

}

}

Near the top of main(), parse the command line to determine if the -echo option is present. Do this like the following:

if(argc > 1 && strcmp("-echo",argv[1])==0) {

ECHO=1;

}

Finally, at each point you read input from the user, immediately echo it back. Echos should include newlines for readability so lines like the following will be common.

fscanf(stdin,"%s",buf);

echo(" "); echo(buf); echo("\n");

It will take some work to get the exact placement of echoes correct but will ultimately lead to nice results that involve LITTLE typing like the example below.

> cat data/stranger-demo.script | ./tree\_main -echo

BST Demo

Commands:

print: shows contents of the tree in sorted order

show: shows contents in pre-order indicating its structure

clear: eliminates all elements from the tree

quit: exit the program

add <name>: inserts the given string into the tree, duplicates ignored

find <name>: prints FOUND if the name is in the tree, NOT FOUND otherwise

save <file>: writes the contents of the tree in post order to the given file

load <file>: clears the current tree and loads the one in the given file

BST> add Lucas

BST> add Mike

BST> add Dustin

BST> add Will

BST> add El

BST> print

Dustin

El

Lucas

Mike

Will

BST> show

Lucas

Dustin

El

Mike

Will

BST> find Mike

FOUND

BST> find Nancy

NOT FOUND

BST> add Nancy

BST> show

Lucas

Dustin

El

Mike

Will

Nancy

BST> find Max

NOT FOUND

BST> add Max

BST> show

Lucas

Dustin

El

Mike

Max

Will

Nancy

BST> print

Dustin

El

Lucas

Max

Mike

Nancy

Will

BST> find Barb

NOT FOUND

BST> add Barb

BST> show

Lucas

Dustin

Barb

El

Mike

Max

Will

Nancy

BST> save stranger.tree

BST> clear

BST> print

BST> add Demigorgon

BST> show

Demigorgon

BST> load stranger.tree

BST> show

Lucas

Dustin

Barb

El

Mike

Max

Will

Nancy

BST> add Jim

BST> add Joyce

BST> show

Lucas

Dustin

Barb

El

Jim

Joyce

Mike

Max

Will

Nancy

BST> load stranger.tree

BST> show

Lucas

Dustin

Barb

El

Mike

Max

Will

Nancy

BST> quit

>

### 5.4 Manual Inspection Criteria for tree\_funcs.c and tree\_main.c

The following criteria will be checked in this problem.

| **Weight** | **Criteria** |
| --- | --- |
| 10 | tree\_funcs.c |
|  | Clear use of binary search principle: left for less, right for greater, use of string functions for this |
|  | Effective use of recursion in node helper methods |
|  | Relatively short functions: nothing needs to be too long (40 lines tops) |
|  | Correct order of visiting/freeing in node\_remove\_all() |
|  | Tree cleared prior to load in bst\_load() to prevent memory leaks |
|  | File closed after finished during saving and loading |
| 10 | tree\_main.c |
|  | Clear structure of main loop, reading commands, selecting actions |
|  | Use of string functions to identify which command to execute |
|  | Easy to recognize reading of additional arguments for add/find/etc. |
|  | Clear efforts to honor -echo option and echo commands |
|  | Clear effort to free all memory prior to exit by clearing tree on exit |

### 5.5 Tests for Tree Functions   GRADING

| **Weight** | **Criteria** |
| --- | --- |
| 15 | test\_tree\_main.sh |
|  | 15 tests for tree\_main executable, exercises functions in tree\_funcs.c |
|  | Compile and run using make test-p3 OR |
|  | Run script using ./test\_tree\_main.sh |
|  | 1 point per test passed |
|  | NOTE: individual tests can be run by doing |
|  | ./test\_tree\_main.sh 4 # runs test 4 |
| 5 | Full credit if Valgrind in test\_tree\_main.sh does not find memory problems |
|  | Deductions based on severity of memory problems identified |

The test\_tree\_main.sh script creates outputs in the test-data/ directory such as test-data/actual.txt which is the output on test which may be useful to inspect. Running a single test as in

./test\_tree\_main.sh 4

ensures that files in the test-data/ directory correspond to the results of that single test.

## 6 Zip and Submit

### 6.1 Submit to Canvas

Once your are confident your code is working, you are ready to submit. Ensure your folder has all of the required files. Create a zip archive of your lab folder and submit it to Canvas.

On Canvas:

* Click on the *Assignments* section
* Click on the appropriate link for this lab/assignment
* Scroll down to "Attach a File"
* Click "Browse My Computer"
* Select you Zip file and press OK

### 6.2 Late Policies

You may wish to review the policy on late project submission which will cost you late tokens to submit late or credit if you run out of tokens. **No projects will be accepted more than 48 hours after the deadline.**

<http://www-users.cs.umn.edu/~kauffman/2021/syllabus.html#late-projects>