MREN 178 - Lab 3

Binary Search Trees

Jano Fu

Michael Kurdyla

Matthew Pan, PhD

A. Introduction

A.1 Objectives

The goal of this lab is to experiment with accessing serial port directly without using Arduino IDE, but using a C program instead. You will be reading data collected from a ultrasonic sensor that measures the distance between an object and itself (aka rangefinding). With the data collected, you will be constructing a Binary Search Tree using the collected measurements.

In this lab, we will be using the HC-SR04 distance-measuring ultrasonic sensor. This economical sensor provides 2 to 400 cm of non-contact measurement functionality with a best-case resolution of 3mm. The sensor emits a current signal less than 2mA.

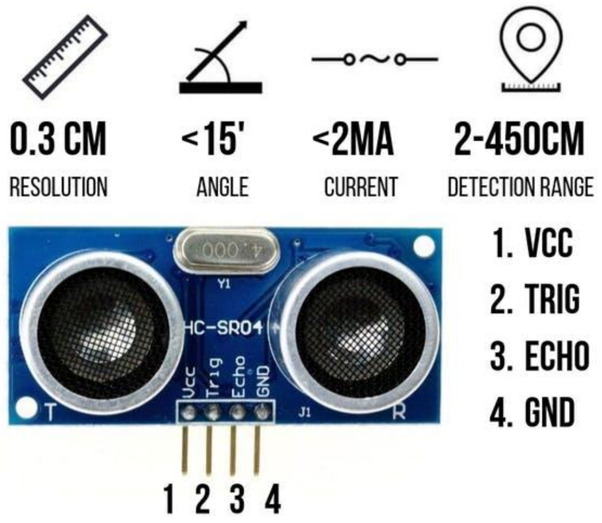


Figure 1. Specifications and photo of the HC-SR04 ultrasonic distance sensor.

To learn more about the sensor, you can review the datasheet here:

https://cdn.sparkfun.com/datasheets/Sensors/Proximity/HCSR04.pdf

A.2 New Topics Covered

- HC-SR04 Rangefinding Sensor
- Binary Search Tree

A.3 Pre-Requisite Knowledge

This lab assumes that you have completed the pre-lab for Lab 3 as you will use the examples as a reference for completing Lab 3's deliverables and have reviewed the general concepts/method of Binary search tree. You will also be using C programming knowledge for this lab exercise, using a C compiler of your choice.

A.4 Equipment

Most everything that you need to complete this lab is included.

From your MREN 178 lab kit, you will need:

- Arduino UNO R3 board
- HC-SR04 Ultrasonic sensor
- Female-Male Jumper Wires
- Male USB-A to Male USB-B Cable

From the OnQ MREN 178 course site, you will need:

- Lab 3 Instructions (this document)
- Pre-Lab 3 Instructions and code (for reference)
- A zip file containing source code for the lab.

On your computer, you'll need:

- A C programming IDE (e.g., CLion, VSCode, etc.)
- Internet Access

⚠

For this lab, you will need to use a Windows-based computer as the serial communications source code used depends on Windows-specific headers.

B. Lab Setup

B.1 Connecting the Arduino UNO R3 and HC-SR04 board

In Lab 3, you will not be using the LCD keypad shield. Thus, if you still have the shield mounted on the Arduino, you can remove it and repackage it in your kit using the Styrofoam (to protect the pins) and the anti-ESD bag. Connect the HC-SR04 ultrasonic sensor to the Arduino using the following wiring diagram:

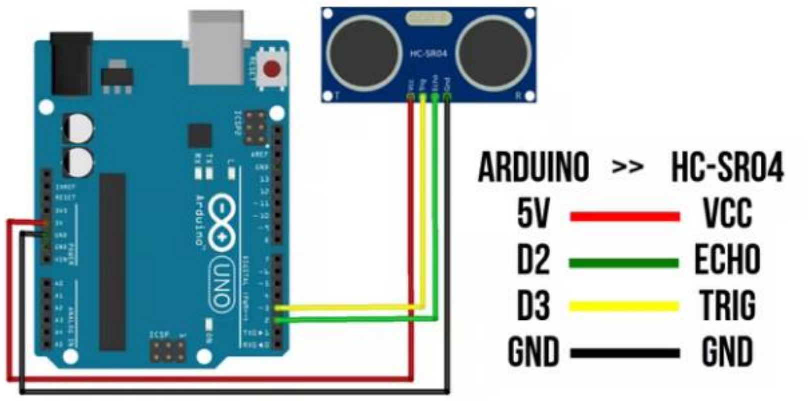


Figure 2. Wiring diagram for Lab 3 - connecting the HC-SR04 sensor to the Arduino UNO.

All we need to do is connect the 4 pins of HC-SR04 direcly to the Arduino using Female-Male jumper wires. Alternatively, you can also use the solderless breadboard and Male-Male jumper wires to ensure that the sensor is upright when measuring distances.

B.2 Loading Arduino Code

In this lab, the Arduino will act as a stand-alone data generator – the sole function of the Arduino is to read data from the HC-SR04 ultrasonic sensor and output it to serial at 9600 baud.

Reading data from serial means accessing data that are printed into the Serial port through the command “Serial.print()”. To have the Arduino correctly reading the sensor data, we first need to upload the code.

- go into the folder “ArduinoCode” and open the folder “UltrasonicSensorBuild”, then open the file “UltrasonicSensorBuild.ino”.
- Connect the HC-SR04 sensor to the Arduino (Figure 2).
- Connect the Arduino to your computer using the USB cable.
- Finding the correct board and port of your connection.
- Upload the code to the Arduino.
- Read through the code to understand what’s happening.

The code itself should be pretty easy to understand. If you have difficulties understanding the code, read the HC-SR04 data sheet (link found above), or ask a TA.

Once the code is uploaded, open up the Arduino Serial Monitor to see what gets outputted (ensure that the communication rate selected is 9600 Baud). The distances outputted are in centimeters (as a float). To test your setup, you can use a ruler and an object placed in front of the ultrasonic sensor (see Figure 3) at a specified distance and check if the output on the serial monitor.

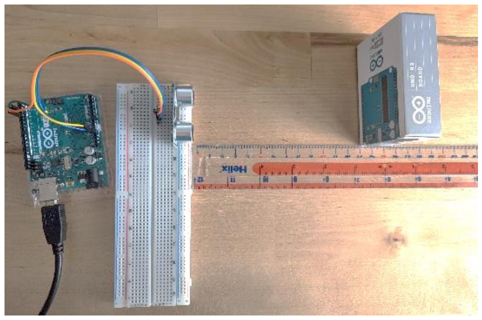


Figure 3. Example test setup for Lab 3.

Most likely, the reading given by the sensor will be within 1 cm of accuracy. You can choose to correct this offset directly in the Arduino code if you wish. Additionally note that this program has a delay of 1 second between sensor reads (i.e., `delay(1000)`). This can be changed to something else if you want faster or slower incoming data.

⚠

Make a note of the COM port that the Arduino uses. This will be needed as part of the first lab exercise below.

Once you've read over the Arduino code and understand what's happening, make a note of the COM port that the Arduino is on and then you can close the Arduino IDE – we won't be needing to modify the Arduino .ino code for the remainder of this lab.

C. Lab Exercises

For the exercises in this lab, you'll need to use a C programming IDE and compiler of your choice. Because this project contains multiple .c files, you'll need to ensure that your compiler is set up to compile and link all source code contained in the folder. Thus, see instructions in Appendix A.

C.1 First BST Program—Storing Measured Data using BST

Description

The code for this part of the lab is found in the folder 'lab03_1'. There will be a few source code files inside:

- *lab3_bst.c* – This .c file contains the implementation of the binary search tree and associated algorithms. Some of these functions are incomplete, and will require you to fill it in.
- *lab3_bst.h* – This is the associated header file for lab3_bst.c. It contains function prototypes and struct definitions.
- *lab3_main.c* – This is where the main function is. You may need to change some the `#define`'ed globals such as COMPORT (the COM port the Arduino is on – you can obtain this from the Arduino IDE), DATAREADGAP (default is 1000 = 1 sec, should be aligned with delay in the .ino file)
- *SerialClass.c* – The functions in this file are used to read incoming data on the serial port serial port
- *SerialClass.h* – the associated header file for SerialClass.c

Again, you should look over all files to ensure that you understand how the program functions. In this part of the lab, you will be completing the code to implement three core functions of a simple BST tree.

The code snippets to be completed are:

- `height()` is used to calculate the height of a binary tree. It takes the root node of the tree, returning the height of the current tree. Note that the height of an empty tree is -1, the height of a leaf node is 0, and the height of other nodes is 1 more than larger height of node's two subtrees.
- `findParentHelper()` finds the parent nodes of the node with a given "key"(the value that node is assigned). It takes in a node that has at least one child. This function is to be used by findParent(), which can be used to return the parent of a node whose key is equal to k.
- `delete (Node *p, Node *n)` is expected to delete the node pointed by n, we have p pointing to the parent of the node that is to be deleted.

Once complete, the code will read the 10 serial data points from the Arduino on the specified COM port (by default, one data point a second). You'll need to quickly adjust the distance of an object from the ultrasonic sensor between sensor reads, and the code will construct a BST from the data.

Deliverables

lab3_bst.c

- Completed `height()`, `findParentHelper()`, and `delete()` functions that allow for the proper execution of the program as described above.

C.2 Second BST Program— Large Data Volume Storage

Description

For this section, you will be implementing a binary tree yourself. Create a file named lab3_2.c in the lab3_2 folder.

You have been given a large file, IDENTS.txt, that contains lines (or records) with numbers and strings of random characters. (This file has been output by the program MAKEIDNT. You can review that code to see how the file was generated.)

(As an aside, you should think about testing as you think about program requirements. A suggestion is that you create small versions of the data – perhaps 10 or 20 lines – and use those for testing. That way, you can debug your program using small samples before you try using the large file.)

The numbers can be thought of as account numbers or employee id numbers, and the strings as encoded passwords. The idea is, we want to store all this information in a BST, so that we can do rapid lookups of passwords, given an identification number.

As you can confirm, the identifications were generated randomly, with no attempt made to make sure every number in the file is unique. In fact, you will see that MAKIDNT has deliberate code to create duplicates. So, the requirement is to treat the second and subsequent occurrences of the same number as updates to the password stored for that number. For example, suppose one of the numbers was 12345678. Suppose also that there are 3 lines with the number 12345678 – the first with string ABCD, the second with string EFGH and the third with string JKLM. Upon encountering the first line with 12345678, your program will find the place to insert the data and will store the number along with the string ABCD. When the second line with 12345678 is found, your program will discover that there is a node in your tree with that key (number) and instead of rejecting the data, will simply update the string to EFGH. Similarly, the third time 12345678 is found to be already present, the password will be changed to JKLM.

The amount of data is such that it would be difficult to determine by human inspection whether the tree is a valid BST. So, one of the requirements is that you write a function that, given a pointer to the root of a BST, determines whether the BST meets the ordering requirements. You may design your routine to work based on the definition of a BST, or perhaps you could modify or build on one of the existing traversal routines to satisfy this requirement. This function should output the in-order traversal of the BST.

There is a second file – DELETES.txt – that contains just numbers. The numbers in this file are identifications that are to be removed from your BST.

There is a third file – LOOKUPS.txt that just contains numbers. The numbers in this file will be used in a test of your BST.

It is highly recommended that you review BST_STRINGS.c and the code in the previous two labs for ideas on how to write this program.

Deliverables
lab3_2.c

- Read in the data from IDENTS.txt and using the data, create a BST
  - Update any nodes with repeating identifications
  - Once all the data has been read in, output the number of nodes in the BST in the form “BST NODES: 12345”
- Read in the data from DELETES.txt and remove the selected nodes from the BST
  - Once all the data has been read in, output the number of nodes in the BST in the form ”NODES AFTER DELETION: 12300”
- Read in the data from LOOKUPS.txt and search the BST for the requested Employee ID Number. If the number is found, output the Employee ID Number and password, otherwise output that the Employee ID Number is not valid, and give the Employee Number. For example, “Employee ID 12345678 is not valid” .
- Create a function that traverses the BST in-order and outputs the Employee ID Numbers of the traversal

D. Lab 3 Deliverables and Grading Rubric

Component	Item	Mark	Max
Lab3_1	lab3_bst.c		17
	`height()`		7
	`findParentHelper()`		7
	`delete()`		3
Lab3_2	lab3_2.c		28
	`Binary Search Tree implementation`		15
	`Update`		2
	`Delete`		4
	`Lookup`		2
	`In-order Validation`		5
	Total		45

Appendix A – Compiling C Projects Containing Multiple .C Source Files

This may be your first encounter with C-based projects that have multiple .c and .h files. This is fairly typical of larger coding projects where you have multiple files that will need to be compiled and lined together. Here, we'll go over some methods for allowing typical compilers you may be using (VSCode or CLion) to compile all of the source code at once, and link them into a single executable file.

VSCode with MinGW

If you have VSCode installed along with the MinGW tools (see Introduction to Visual Studio Code document found in MREN 178 OnQ > Content > Resources), you can compile all source files within a folder using a simple command in the terminal.

1. To open a terminal in VSCode, use the hotkey Ctrl+Shift+' or go to Terminal->New Terminal. A new terminal with command prompt should open up (by default at the bottom of the screen). Next, ensure that the command prompt has the path to the source code you wish to compile followed by the '>' sign, something like this:

   ```
   C:\Courses\MREN 178\Labs\lab3\lab03_1>
   ```

   This ensures that whatever command you enter at the prompt will execute in the named folder/path. If the correct folder isn't listed, use the commands 'dir' and 'cd' (stands for change directory) to view contents and change the directory if you're using Windows (for MacOS and Linux, use 'ls' and 'cd'). Use `cd ..` (two dots) to go back one folder, and `cd [folder_name_here]` to go to a folder with the name [folder_name_here] that's in the current directory.

2. Type the following command at the prompt once you're in the correct folder:

   ```
   gcc *.c -o [exe_filename]
   ```

   This command tells your computer to run gcc (the GNU C Compiler) on all (that's what the * represents) files in the folder with the file extension `.c,` and outputs `(-o)` the executable using the name [exe_filename].

3. After pressing enter, you'll get an executable with the name you specified in the folder (or you might get compile errors if there's something wrong).

4. Type the filename of the newly generated executable to run your program.

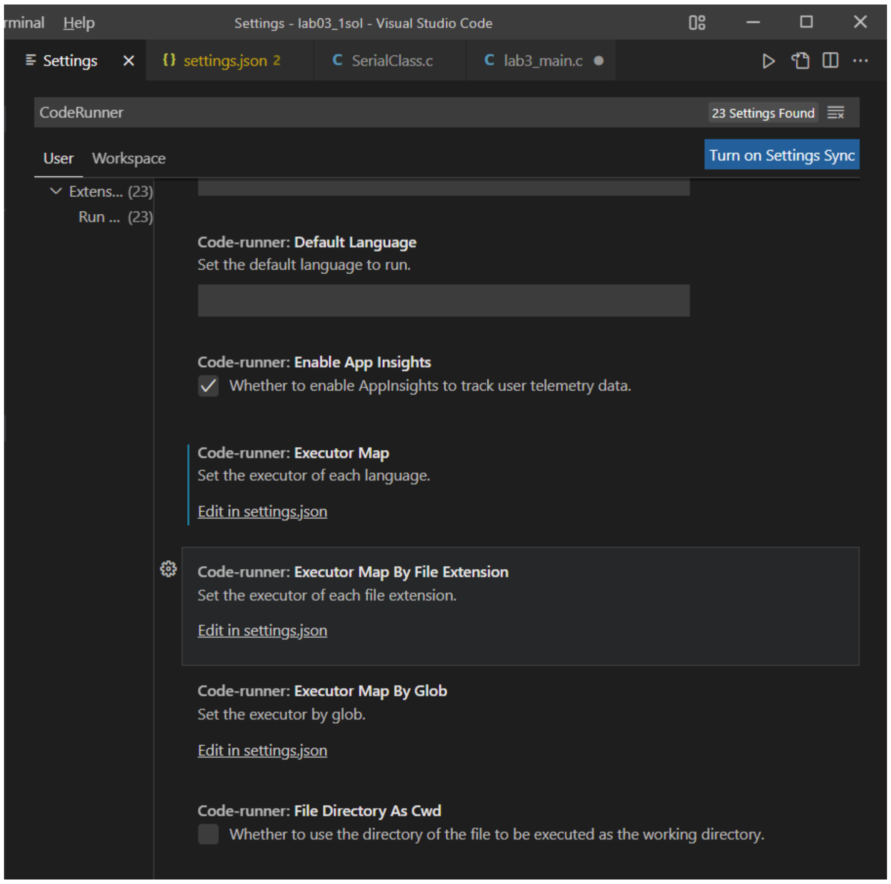
VSCode with MinGW and CodeRunner

If you use the CodeRunner extension to compile and run your code by pressing the hotkey Ctrl+Alt+N, then you may likely get a bunch of undefined reference errors when using that hotkey on a C project with multiple .c and .h files in the folder. That's because CodeRunner is only set up to compile one .c file at a time by default. To change this behaviour, we're going to need to change CodeRunner's settings.

1. Open your user and workspace settings using following VS Code menu command:
   a. On Windows/Linux - File > Preferences > Settings
   b. On macOS - Code > Preferences > Settings

   Alternatively, use the hotkey Ctrl+'

2. Type 'code-runner' in the Search Settings bar and scroll down to the option 'Code-runner: Executor Map By File Extension" and click on 'Edit in settings.json'



Code-runner: **Default Language**
Set the default language to run.

Code-runner: **Enable App Insights**
✓ Whether to enable AppInsights to track user telemetry data.

Code-runner: **Executor Map**
Set the executor of each language.
Edit in settings.json

Code-runner: **Executor Map By File Extension**
Set the executor of each file extension.
Edit in settings.json

Code-runner: **Executor Map By Glob**
Set the executor by glob.
Edit in settings.json

Code-runner: **File Directory As Cwd**
Whether to use the directory of the file to be executed as the working directory.

3. A new VSCode tab will appear with some JSON script. In the script, find the line with the text `"code-runner.executorMap"` and within its brackets, paste the following line:

```
"c": "cd $dir && gcc *.c -o $fileNameWithoutExt && $dir$fileNameWithoutExt",
```

If you are entering this line at the end of the mapping list, you need to add a comma at the end of the previous item in the list, and remove the comma at the end of the newly inserted line. Thus, your code-runner.executorMap could look something like this:

```
    "code-runner.executorMap": {
        "c": "cd $dir && gcc *.c -o $fileNameWithoutExt && 
$dir$fileNameWithoutExt",
        "cpp": "cd $dir && g++ -std=c++14 $fileName -o $fileNameWithoutExt && 
$dir$fileNameWithoutExt"
    }
```

If the JSON file doesn't include a code-runner.executorMap field, then copy the code immediately above and paste it at the end of the JSON. Ensure that there is a comma at the end of the previous line prior to code-runner.executorMap. Here's an example of what my full settings.json looks like:

```
{
    "terminal.integrated.shell.windows": "C:\\Windows\\System32\\cmd.exe",
    "python.defaultInterpreterPath": "C:\\Users\\Matt\\anaconda3\\python.exe",
    "code-runner.runInTerminal": true,
    "window.zoomLevel": 1,
    "C_Cpp.updateChannel": "Insiders",
    "security.workspace.trust.untrustedFiles": "open",
    "code-runner.executorMap": {
        "c": "cd $dir && gcc *.c -o $fileNameWithoutExt && 
$dir$fileNameWithoutExt",
        "cpp": "cd $dir && g++ -std=c++14 $fileName -o $fileNameWithoutExt && 
$dir$fileNameWithoutExt"
    }
}
```

4. Save and close the settings.json file.
5. Now, in your folder containing all of your source code, run 'Ctrl+Alt+N' to compile and run your code with code runner.

CLion

Unfortunately, I have never used CLion. However, from what I've read, CLion uses a build system called CMake. Therefore, in your project folder, there should be a file named 'CMakeLists.txt' that contains instructions for the build system to build your project.

Inside that txt file, you'll either see or need to add the command:

```
add_executable(program file1.c file2.c)
```

This command tells the build system to create an executable 'program' with files 'file1.c' and 'file2.c'. Thus, for a project, say, lab3_1, you'll need to type in the following command in CMakeLists.txt:

```
add_executable(program lab3_main.c lab3_bst.c SerialClass.c)
```

You don't need to type in header files in this command because they will automatically be included by the preprocessor.