ELEC 278 – Data Structures and Algorithms

Lab 3: Binary Search Trees

Start: October 19, 20202 Due: Day 7 of Lab Day

Contents

1. Basic Lab Information

You should review the Lab Overview document prior to reading the material for this lab exercise.

If you have already installed a C compiler on your computer (or have one available), then you can continue with the lab. If you have not installed the C programming tools, please stop working on this lab, and review Lab Overview document Appendix A. You may also wish to review Tutorial 1 before you attempt this lab.

You should also review Appendix B of the Lab Overview document. This contains important information about how to submit your assignment.

You may also find it useful to review Labs 1 and 2 as you work on this lab exercise.

1.1 Background

A **tree** data structure is a collection of 0 or more nodes, where each node contains data, and pointers to subtrees. (Referring to trees with 0 nodes will be helpful when we think about recursive routines – we are done when we find an empty tree.) There is one node, called the **root node**, which is the single node at the top of the tree. All nodes in the tree only have one or zero nodes pointing to them – zero only when the node is the root node. If a node points to another node, it is called the **parent** of that node. The nodes that a parent node points to are called the **children** of that node. Sometimes the terms **grandparent** and **grandchild** will be used to describe the relationship between nodes. A node with no children is called a **leaf**.

When a tree is drawn, lines are used to show the connection from a tree node to the child node(s). These lines are called **branches**. If we consider the root to be at level 0, then the root's children are at level 1, the grandchildren are at level 2, and so on. Following a path from the root through descendants (without backtracking) will go across some number of branches. The maximum number of branches that would be crossed to get from the root to any leaf in the tree is the tree's **height**.

A **binary tree** is a tree where each node can have a maximum of two subtrees (or two children). These are often referred to as the left and the right subtrees (children).

If a binary tree is ***ordered***, then for every node, all the values in the left subtree of that node are less than the value in the node, and all the values in the right subtree of that node are greater than the value in the node. An ordered binary tree is suitable for efficient searching. That is, if a particular value is being sought, either it is found in a particular node, or if not, then ***only the left or right subtree needs to be searched***, based on whether the value is less than or greater than the value in the node. An ordered binary tree is called a **binary search tree**. (Note that the ordering could be opposite to what was described above – with larger values on the left and smaller values on the right. Everything would work the same way, except that the comparisons would be opposite.)

The diagrams later in the lab material will be helpful to visualize binary trees.

2. Lab Resources

Specific to this lab:

1. Lab Instructions (this document)
2. A zip file containing source code (in a folder called Lab03Src). Check the file "PACKING.LIST" (it is a plain text file) for a list of files that are included.

Learner provided:

1. Programming environment selected by the learner. The Lab Overview document provides a list of possible C programming tools that you may use.

This lab has code for you to read. The purpose of the example code is to reinforce what you have seen in the class material. It also provides you with code to cut-and-paste into the code you write to solve the assigned problems. This lab has multiple programming problems to solve.

2.1 List of Files provided with this Lab

- BST_STRINGS.c
- BST_TRAVERSAL.c
- DELETES.TXT
- IDENTS.TXT
- LAB3_BST.c
- LAB3_BST.h
- LAB3_MAIN.c
- LOOKUPS.TXT
- makefile
- MAKEIDNT.c
- PackingList.txt
- TEST.ini

(See the file PACKING.LIST included in the Lab03Src.zip file for the most up-to-date list.)

3. Lab Objective

There are three objectives for lab 3:

a) Review example code that (partially) implement binary search trees.
b) Add code to an existing BST program to complete the required functionality.
c) To design a new BST program, using provided code as a model, and to test the code.

3.1 Topics covered in this lab

- Binary Search Trees (BSTs) – Implementation choices.

4. Lab Steps

There are three main steps in this lab. The first step is a reading or study steps – you have some code that illustrates programming concepts and you are to develop an understanding as to how the code works. The last two step involve work you must complete – by using materials that you studied in the earlier parts. Some of you may find it easier if you tackle these steps bit-by-bit. You can nibble on pieces over the two-week period – or have a large feast the night before the lab is due. Figure out which technique works for you. (Just a suggestion – gorging on a large feast in the evening may give you a stomach cramp the next day!)

4.1 Part 1 – Sample Programs Review

BST_TRAVERSAL.c illustrates the three methods of depth-first binary tree traversal. The program has two examples of tree data "hard-coded" – that is, it does not depend on reading data from a source. The important thing to note is the order in which data is printed – depending on whether the traversal is pre-order, in-order, or post-order. If the tree is a binary search tree, the in-order traversal will print the data in sorted order.

BST_STRINGS.c shows an example where the key and the data are non-numeric. In the case of this example, both the key and the data are strings. Note that the tree node data structure does not store the whole string; it stores pointers to the key string and the data string. The program uses a modified version of the tree traversals in the first program to print just the key fields of the nodes in the tree. This piece of code will serve as a starting point for the second deliverable.

4.2 Part 2 – Deliverable 1 – First BST program

There are three files that start with the name LAB3 – LAB3_BST.h, LAB3_BST.c and LAB3_MAIN.c. These are parts of a BST program. There are pieces missing in LAB3_BST.c, and the goal of this part of your work is to add those pieces. There are many comments in the code which you will find useful. Note that it is only LAB3_BST.c that has parts to be added. There are some minor changes to be made in LAB3_MAIN.c, to test your code as parts are completed.

4.2.1 Add height() function

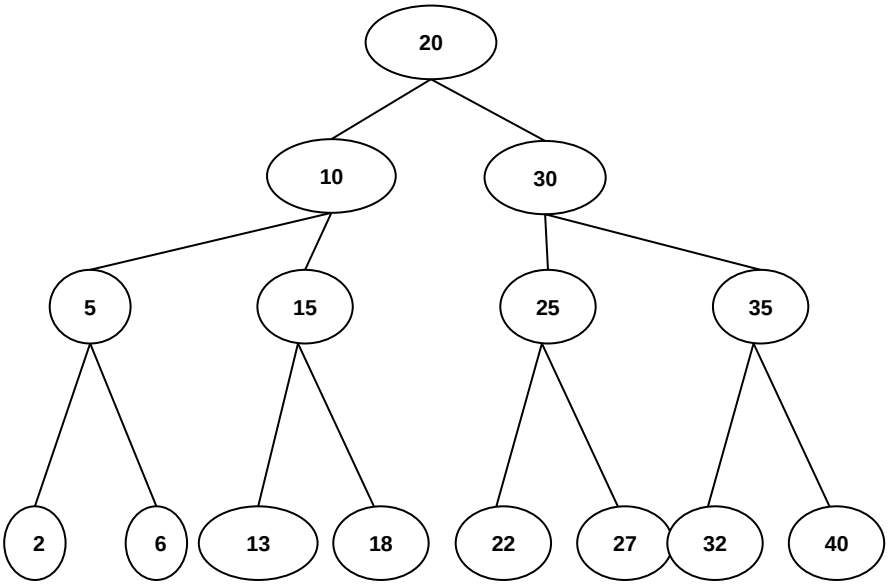
You will implement a function to calculate the height of a binary tree. The skeleton for the routine height() is provided in the LAB3_BST.c code.
It is suggested that you write the code for this requirement first and test it before moving to the second step. It is further suggested that you do the same for steps 2 and 3.

4.2.2 Add findParentHelper() function

You will implement a function to find the parent of node.
In this task, you will implement the following function in **`bst.c`** :

```
Node* findParentHelper(Key k, Node* root)
```

which is a helper function called upon by function **`findParent().`** Its job is to return the parent of a node when the root has at least one child. Once implemented correctly, **`findParent()`** can be used to return the parent of a node whose **`key`** is equal to **`k.`** For example, for the following tree:



if we call **`findParent(15, tree->root)`** we should get the node with **`key = 10`** returned.

Hint 1: Look at the **`Node *find( Key k, Node *root)`** implementation.
Hint2: Need to check if either child of a node has a **`key`** equal to **`k`**.

4.2.3 Add delete() function

You will implement a function to delete a node from the tree. In this part, you are required to implement the **`delete`** function in **`tree.c.`** It gets called by the **`withdraw`** function. The delete function has the following signature:

```
void delete (Node* p, Node *n);
```

The **`delete`** function takes in two nodes, **`p`** and **`n,`** where **`p`** is the parent of node **`n.`** **`withdraw`** has already been implemented.

Hint: Refer to lecture slides to refresh your memory on the different cases that you must deal with when removing a node from a binary tree.

Detailed specifications:

- The source file name that you provide shall be LAB3_BST.c. (The other files are unchanged and do not need to be submitted.)
- The program does not take input – the values to insert are hardcoded in LAB3_MAIN.c. When testing your program, the values used by the markers in the test code may be different.
- The output is determined by the code in LAB3_MAIN.c – ***PLEASE DO NOT ALTER THE OUTPUT FORMAT***.

- The first line of your program shall conform to the course standard for headlines, that is, it should be
  // LAB3_BST.c – Lab 03 – FirstName, LastName

4.3 Part 3 – Deliverable 2 – Second BST program

BACKGROUND

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.

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.

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.

You should review BST_STRINGS.c and the code in the previous two labs for ideas on how to write this

program.

REQUIREMENT

The program shall:

1) Read in the data in file IDENTS.txt and create a BST. The program shall report the number of nodes in the BST created by printing BST NODES: 12345 where the number is the actual number of nodes in your BST.
2) Using the function that tests for a valid BST, your program will determine that the BST is valid. If it is not, your program shall terminate, and you shall determine what the problem is and correct it.
3) Read in the file DELETES.txt and use that data to remove some of the nodes in your BST. The program shall report the number of nodes in the BST by printing NODES AFTER DELETES: 12010 where the number is the actual number of nodes in your BST.
4) Using the function that tests for a valid BST, your program will determine that the BST is valid. If it is not, your program shall terminate, and you shall determine what the problem is and correct it.
5) Finally, read in the third file, LOOKUPS.txt and use that data to perform searches. For each number in the file, your program shall look up the node with that number and shall print the following information: ID 12345678 PASSWORD XXXXXXXX
   If the ID is not found, the password value will be "<NOT FOUND>" (just the string between the quotes).

Detailed specifications:

- The source file name shall be LAB03BST2.c.
- The input shall be files with the names as specified above in the program requirements.
- The output shall be as specified in the program requirements.
- The first line of your program shall conform to the course standard for headlines, that is, it should be
  // LAB03BST2.c – Lab 03 – FirstName, LastName