ELEC278

Lab 3 - 2019

Search Trees

Contents

1. Objective

The purpose of this lab is to give you some practice writing recursive search tree methods.

2. Background

A **tree** data structure is a collection of nodes, where each node contains data and pointers to subtrees. There is one node, called the **root 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 particular node. The nodes that a parent node points to are called the **children** of that node.

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 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 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.

3. Lab Organization

This lab includes three steps, all of which involve adding code to the binary search tree code provided to you in the folder found in archive **`lab3src.zip`**

In Section 5, you will implement a function to calculate the height of a binary tree.

In Section 6, you will implement a function to find the parent of node.

In Section 7, you will implement a function to delete a node from the tree.

Section 8 offers the opportunity to consider better ways to implement the code for a binary search tree.

All of the function will be tested using the code in **`main.c`**. After solving all of the problems, show your programs to a TA in order to get your mark for this lab. You are allowed to work in groups of a maximum of two people. The only changes you can make to the code in **`main.c`** is uncommenting lines to test your code.

4. Provided Binary Search Tree Code

In this lab you will be making use of the provided binary search tree code (**`bst.h`** and **`bst.c`**). The particular binary search tree you will use is the same as was presented in the lecture slides:

```
typedef int Key;

typedef struct Node {
     Key   key;
     void  *value;
     struct Node *leftChild, *rightChild;
} Node ;

typedef struct Tree
{
     Node *root;
} Tree ;
```

The following function have already been implemented:

```
Node *initNode (Key k, void *v);

Tree *initTree(Key k, void *v);

Node *find( Key k, Node *root);

int insert(Key k, void *v, Node *root);

int max(int a, int b);

Node* findParent(Key k, Node* root);

Node* widthdraw(Key k, Node* root);
```

You have seen most of these functions in class. The latter two functions, **`findParent() and withdraw()`**, call on helper functions that you will implement. If you are unsure about the functionality of any of the functions, refer to the comments in the code for clarification. The functions that you will implement are:

```
int height(Node *root);

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

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

The code in main.c has three pieces of code bracketed by #if directives. Those pieces of code will be included if the corresponding manifest is defined, as shown in the following lines.

```
#define     HEIGHT_WRITTEN      0
#define     FINDPARENTHELPER_WRITTEN      0
#define     DELETE_WRITTEN      0
```

In Part 5 (Step 1), you will write the height() function. To test it, you will change the line in main.c to

```
#define     HEIGHT_WRITTEN      1
```

When you do this, the code between the brackets gets included when the program is compiled, and code to test your height() function will execute when your program is run.

The other two manifests are used in a similar manner for step 2 and step 3.

4.1 Task: Get, Build and Test Program

4.1.1 Get Lab 3

Download **lab3src.zip** from the OnQ course page and unzip the folder.

4.2 Task: Build and Run the Program

Type in the command:

```
cc bst.c main.c –o mytree
```

Run it:

```
./mytree
```

If the program runs successfully, your output should look like this:

```
Original Tree:
{(10,1),{(3,1),{(1,1),{},{}},{(7,1),{},{}}},{(20,1)
,{(15,1),{},{(18,1),{(17,1),{(16,1),{},{}},{}},{}}}
,{}}}
```

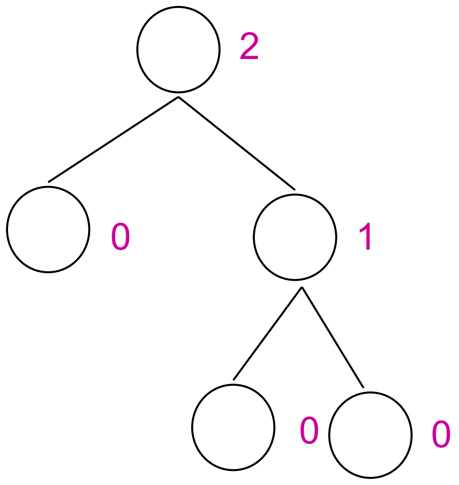
```
Height of tree: 66544 (Note: Your value will be
different)
```

4.2 Deliverable

None.

5. Step 1: Calculating the Height of a Binary Tree

The height of a tree is the length of the **longest** path from the root node to a leaf of the tree. The height of a leaf is zero and it increases as you go up the tree. Here is an example of a binary tree with height 2:



5.1 Task

Your task is to implement the function: **`int height(Node *root)`** which calculates the height of the tree. Write your implementation in **`bst.c`** where indicated.

Hint 1: Use Recursion. Remember every subtree is a tree.
Hint 2: Make use of the provided **`int max(int a, int b)`** function.

Hint 3: It is always helpful to make sure you can explain how your code will get to a solution BEFORE you start writing code.

5.2 Deliverable

1) Recompile your code:
   `cc bst.c main.c –o mytree`
2) Show the output of running **`./mytree`**, as well as your source code, to a TA before the end of the lab.

6. Step 2: Finding the parent of a node

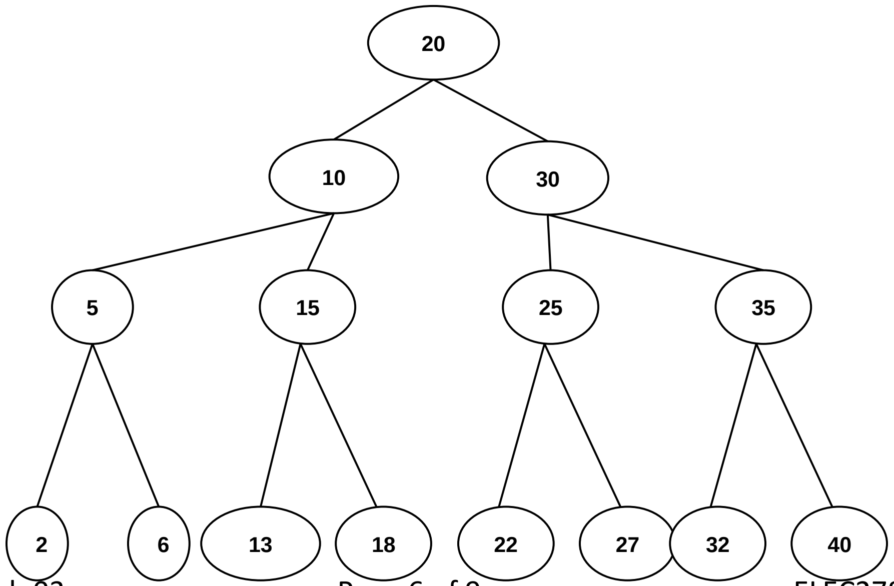
Since our tree **`struct`** doesn't include a pointer to the node's parent, which we will need to implement the **`delete`** operation, we need a function to find the parent of a node.

6.1 Task

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 children of a node have a **`key`** equal to **`k`**

6.2 Deliverable

1) Redefine the appropriate manifest to select this section of the lab.
2) Recompile your code
3) Show the output of running **`./mytree`** as well as your source code, to a TA at before the end of the lab.

7. Step 3: Deleting a node from the tree

Withdrawing a node from the tree first copies its key and value into a brand new node and then it deletes that node from the tree. However the tree must remain structurally sound after the delete operation

7.1 Task

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

7.2 Deliverable

1) Redefine the appropriate manifest
2) Recompile your code
3) Show the output of running **`./mytree`** as well as your source code, to a TA at the end of the lab.

8. Stretch Goal – OPTIONAL. Building an alternate tree implementation.

The code provided with the lab exercise is an implementation of a binary search tree, but it isn't the only version. This optional part of the lab allows the student to look at changes to the code.

An issue to consider is that the implementation requires a first node be present in the tree in order for all the routines to work. (Review the code to see why all the functions as written – including insert – cannot work on an empty tree. Look carefully at how the functions would handle an initial call with no first node.)

Redesign the implementation of a tree, so that a no-node (or null) tree is possible. Implement the insert function to work with your modified scheme, and prove that it works, starting with an empty tree.

8.1 Deliverable

Demonstrate to the TA that your version of the tree structure and your version of the insert using your structure will work.

ELEC 278 LAB EXERCISE 3 CHECK-OFF SHEET

Student Name	
Student Number (8 digits)	

Part Number	**Step Description**	**Mark**	**TA Sign Off**
Step 1	Implement and demonstrate height()	2	
Step 2	Implement and demonstrate FindParentHelper()	4	
Step 3	Implement and demonstrate delete()	4	
8 – Stretch (optional)	Re-implement the structure of the tree, and demonstrate new version of insert(), starting from an empty tree.	3	
Total	Full marks (complete lab requirements)	10	
	Maximum possible marks	13	