CS 010C – Assignment 2
Andrew Pham
Collaborated with Arjun Bhalla, Rushil Shah, and Devesh Panda

Problem 1: (10 points) Inspect the tree shown below and answer the following questions.

1. (1 points) What is the height of the tree?

The height of any node is the number of edges on the longest path from that node to a leaf. Thus, the height of a tree is the number of edges on the longest path from the root node to a leaf. The height of the given tree is therefore 3.

2. (1 points) What is the height of the subtree rooted at F?

The height of the subtree rooted at F would be the number of edges on the longest path from the node F to a leaf. The height of the subtree rooted at F would therefore be 1.

3. (1 points) How many siblings does the node E have?

Two nodes are called siblings if they share the same parent node. The node E shares a parent node with the node D; therefore it has 1 sibling.

4. (1 points) List down the ancestors of the node H in a descending order of their depth.

The ancestors of a given node are the parent of that node, and the parent node's parent, and so on. So, the ancestors of H in a descending order of their depth would be: E, B, A.

5. (2 points) Enumerate the tree nodes in the preorder traversal. You do not need to show any intermediate step. Only the final order is required.

preorder: A, B, D, G, E, H, C, F, I

6. (4 points) Repeat the last part for the inorder and the postorder traversals.

inorder: G, D, B, E, H, A, C, F, I
postorder: G, D, H, E, B, I, F, C, A

Problem 2: (10 points) Write down recursive tree algorithms for the following problems. In each problem, the input is a pointer to the root of a binary tree. The definition of the tree node is shown below. Your algorithm should support the special case of an empty tree which is denoted by a NULL pointer to its root. Analyze the worst-case running time of each algorithm using the Big-Oh notation.

1. (2 points) Compute the height of the tree. Assume the height of an empty tree is -1.

Worst-case runtime: O(n)

```
int height(tree_node *root) {
    if(root == nullptr) {
        return -1;
    }
    else {
        int heightLeftChild = height(root->left);
        int heightRightChild = height(root->right);

        if(heightLeftChild > heightRightChild) {
            return 1 + heightLeftChild;
        }
        else {
            return 1 + heightRightChild;
        }
    }
}
```

2. (2 points) Compute the sum of all values in the tree nodes. The sum of an empty tree is 0.

Worst-case runtime: O(n)

```
int sum(tree_node *root) {
    if(root == nullptr) {
        return 0;
    }
    else {
        return root->value + sum(root->left) + sum(root->right);
    }
}
```

3. (2 points) Search for a given integer value x and return true if the tree has a node with the given value; false otherwise.

Worst-case runtime: O(n)

```
bool search(tree_node *root, int x) {
    if(root == nullptr) { return false; }
    if(root->value == x) { return true; }
    return (search(root->left, x) || search(root->right, x));
}
```

4. (4 points) Compute the number of nodes at each level of the tree. The return value is a list of integers where the entry at the position i ∈ [0, h] (h is the height of the tree) indicates the number of nodes with depth i. For an empty tree, an empty list is returned. For example, for the tree used in Question 1, the return value is [1, 2, 3, 3].

Worst-case runtime: O(n^2)

```
//recursive helper function
int count_at_depth(tree_node *root, int currLevel, int target) {
    if(root == nullptr) {
        return 0;
    }
    if(currLevel == target) {
        return 1;
    }
    return count_at_depth(root->left, currLevel+1, target) +
count_at_depth(root->right, currLevel+1, target);
}

/* finds height of tree given the root */
int height(tree_node *root) {
    if(root == nullptr) {
        return -1;
    }
    else {
        int heightLeftChild = height(root->left);
        int heightRightChild = height(root->right);

        if(heightLeftChild > heightRightChild) {
            return 1 + heightLeftChild;
        }
        else {
            return 1 + heightRightChild;
        }
    }
}

/* creates vector using recursive helper containing the number of
nodes at each level of the tree */
vector<int> list_at_depth(tree_node *root) {
    vector<int> v;
    int treeHeight = height(root);
    for(int i = 0; i <= treeHeight; ++i) {
        v.push_back(count_at_depth(root, i, treeHeight));
        cout << v.at(i);
    }
    return v;
}
```

Problem 3: (10 points) Consider the following arithmetic expression:

$$(2 + 3) * 5 + 7 - 10$$

We saw two algorithms in the class, one to convert the infix expression to postfix, and another to evaluate the postfix expression.

1. (5 points) Convert the expression into an equivalent postfix expression using a stack. You should explain each step of your algorithm (i.e., the output and the stack snapshot after each iteration).

Note: ignoring spaces and only caring about operators, operands and parentheses when converting infix to postfix.

i. The first character is an open parenthesis. So we push it to the stack.
ii. The second character is a 2. It is an operand so we append it to our postfix expression.
iii. The third character is a +. It is an operator, so we push it to the stack.
iv. The fourth character is a 3. It is an operand so we append it to our postfix expression.
v. The fifth character is a closing parenthesis. So, we repeatedly append the top of the stack to our postfix expression, then pop the top of the stack, until we reach an opening parenthesis. We pop this parenthesis but we do not append it to our postfix operation.
vi. The sixth character is a *. It is an operator, so we push it to the stack.
vii. The seventh character is a 5. It is an operand so we append it to our postfix expression.
viii. The eighth character is a +. It is an operand but it is of lower precedence than the * at the top of the stack. So we append each character in the stack to our postfix expression and pop them from the stack after appending each one. Then, we push the + to the stack.
ix. The ninth character is a 7. It is an operand so we append it to our postfix expression.
x. The tenth character is a −. There is a +, an operator of the same precedence, at the top of the stack, so we append it to our postfix expression and then pop it, before pushing the - to the stack.
xi. Finally, we have our final operand, 10, which we simply append on to the end of the postfix expression.
xii. Last but not least, we append whatever's left in the stack to the end of the expression, until the stack is empty.

Infix	Stack	Postfix
(2+3)∗5+7−10		
2+3)∗5+7−10	(	
+3)∗5+7−10	(	2
3)∗5+7−10	(+	2
)∗5+7−10	(+	23
∗5+7−10		23+
5+7−10	∗	23+
+7−10	∗	23+5
7−10	+	23+5∗
−10	+	23+5∗7
10	−	23+5∗7+
	−	23+5∗7+10
		23+5∗7+10−

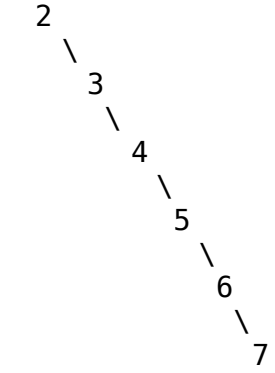
2. (5 points) Evaluate the postfix expression using stack. Here again, you should explain each step of your algorithm.

i. The first character is a 2. It is an operand so we push it to the stack.
ii. The second character is a 3. It is an operand so we push it to the stack.
iii. The third character is a +. So we pop the top two elements (2 and 3) from the stack (storing them in temporary variables before the pop), and then we add those variables together to get 5, which we push to the stack.
iv. The fourth character is a 5. It is an operand so we push it to the stack.
v. The fifth character is a ∗. So we pop the top two elements (5 and 5) from the stack (storing them in temporary variables before the pop), and multiply them together to get 25, which we push to the stack.
vi. The sixth character is a 7. It is an operand so we push it to the stack.
vii. The seventh character is a +. So we pop the top two elements (25 and 7) from the stack (storing them in temporary variables before the pop), and add them together to get 32, which we push to the stack.
viii. The eighth character is a 10. It is an operand so we push it to the stack.
ix. The ninth and final character is a −. So we pop the top two elements (32 and 10) from the stack (storing them in temporary variables before the pop). Subtraction is not commutative, so we take care to subtract the first element we popped (10) from the second element we popped (32). 32 − 10 = 22. We push 22 to the stack.
x. We have reached the end of our postfix expression. Returning the top of the stack gives us our answer, which is 22.

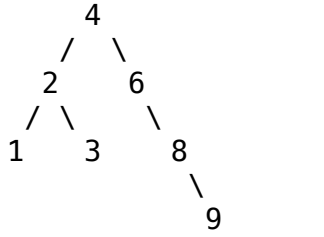
```
Postfix          Stack (operands separated by a space
----------       ------------------------------------
23+5*7+10-
 3+5*7+10-       2
  +5*7+10-       2 3
   5*7+10-       5
    *7+10-       5 5
     7+10-       25
      +10-       25 7
       10-       32
         -       32 10
                 22
```

Problem 4: (10 points) Consider the following set of inputs. For each sequence, construct a binary search tree (i.e., draw the final BST) by inserting the integers from left to right, starting with an empty tree.(5 points)

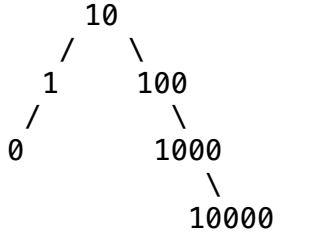
1. 2, 3, 4, 5, 6, 7



2. 4, 2, 6, 3, 8, 9, 1

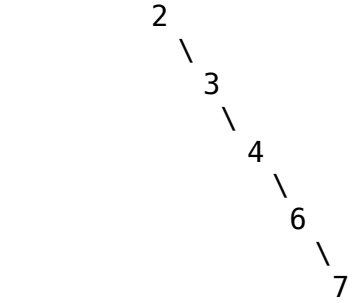


3. 10, 100, 1, 1000, 10000, 0

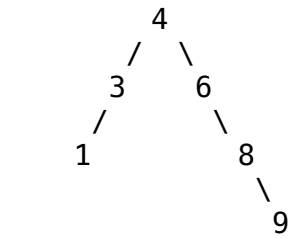


For each of the above sequences, suppose after inserting all the elements, we delete the following elements. Draw the modified BST after performing the above deletion operations.

1. delete 5



2. delete 2



3. delete 10