FIT1008 Introduction to Computer Science (FIT2085 for Engineers)

Tutorial 12

Semester 1, 2019

Objectives of this tutorial

- To understand Binary Trees and Binary Search Trees.

Exercise 1 *

A binary expression tree is a binary tree used to represent algebraic expressions composed of unary and binary operators. The leaves of a binary expression tree are operands, such as constants or variable names, and the other nodes contain operations.

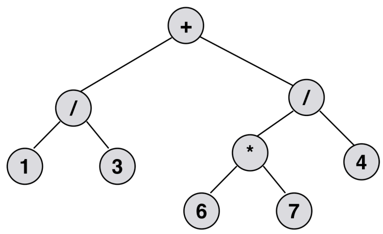


Figure 1: Expression tree

The prefix notation of an algebraic expression results from traversing the corresponding expression tree in pre-order. The infix notation results from traversing the tree in in-order; and the postfix notation (or reverse polish notation) results from traversing the tree in post-order.

Give the unambiguous mathematical expression as well as the prefix, infix and postfix notation of the expression represented by the tree above.

Solution

The expression is $((1/3) + ((6 * 7)/4))$.

- **Prefix:** $+ \ / \ 1 \ 3 \ / \ * \ 6 \ 7 \ 4$
- **Infix:** $1 \ / \ 3 \ + \ 6 \ * \ 7 \ / \ 4$
- **Postfix:** $1 \ 3 \ / \ 6 \ 7 \ * \ 4 \ / \ +$

Exercise 2 *

Consider a `BinaryTree` class which defines a binary tree data type implemented using linked nodes, defined as follows:

```
class TreeNode:
    def __init__(self, new_item=None, left=None, right=None):
            self.item = new_item
            self.left = left
            self.right = right

class BinaryTree:
    def __init__(self):
            self.root = None
```

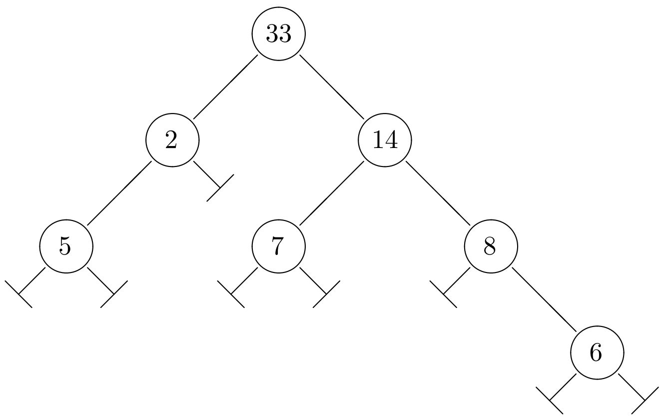
Write down an attribute method for the class that returns the height of the Binary Tree.

Solution

```
def __len__(self):
        return self._aux_len(self.root)

def _aux_len(self, current):
        if current is None:
                return 0
        else:
                leftLen = self._aux_len(current.left)
                rightLen = self._aux_len(current.right)
                return 1 + max(leftLen,rightLen)
```

Exercise 3 *

Add to the class above the method `sum_leaves(self)` which returns 0 if the tree is empty and, otherwise, returns the result of adding the value of every leaf in the tree. For example, for `a_tree` of the form:



the result of `a_tree.sum_leaves()` would be $5 + 7 + 6 = 18$.

Solution

```
def sum_leaves(self):
        return sum_leaves_aux(root);

def sum_leaves_aux(current):
        if current is None:
                return 0
        elif current.left is None and current.right is None:
                return current.item
        else:
                left = sum_leaves_aux(current.left)
                right = sum_leaves_aux(current.right)
                return left+right
```

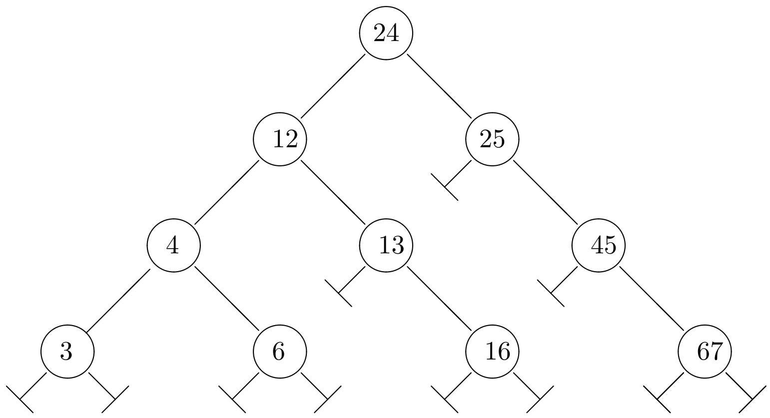
Exercise 4 *

We want to extend the `BinarySearchTree` class defined in the lectures by adding a method `find_min()`. This method returns the minimum key in the tree, or `None` if the tree is empty. In doing so, it does not modify the tree. The following code shows two failed attempts at an implementation of such method:

```
def find_min_1(self):
    return self.find_min_aux_1(self.root)

def find_min_aux_1(self, current):
```

```
    if current is not None:
        return self.find_min_aux_1(current.left)
    else:
        return current

def find_min_2(self):
    if self.root is None:
        return self.root
    else:
        return self.find_min_aux_2(root)

def find_min_aux_2(self, current):
    if current.left is not None:
        return current
    else:
        return self.find_min_aux_2(current.left)
```

Consider a tree `the_tree` with integer keys with the form:



1. Show the value of result after calling `result = the_tree.find_min()` for each definition above.
2. Provide a correct definition for the above method.

Solution

The first definition goes all the way down the leftmost branch and then returns `null`. The problem is that the condition for the if-then-else is `current is not None` rather than `current.left is not None`. Additionally, it returns `current`, rather than the key of the node.

The second definition stops right at the root node (since its left child is not empty) and thus returns a reference to the root node. The problem is that the THEN and ELSE branches of the if-then-else are swapped (or the condition negated). It also suffers from the same problem as the one before: it returns `current` rather than `current.key`

A correct definition is:

```
def find_min(self):
        if self.root is None:
                return None
        else:
                return self.find_min_aux(self.root)

def find_min_aux(self, current):
        if current.left is None:
                return current.key
        else:
```

```
                return find_min_aux(current.left)
```

This method is linear (only one recursive call) and direct (calls itself). It is also tail recursive and can therefore be converted to a simple iteration without the need of a stack.

The value returned by this method for the tree given in the figure would be

Exercise 5

Given a BST with numeric values and two numbers a and b, write down a function that returns a list with all items between a and b. The idea is to do this without visiting all elements, if possible.

Solution

```
def tree_range(self, a, b):
        ans = []
        self._range(a, b, self.root, ans)
        return ans


def _range(self, a, b, current, a_list):
        if current is None:
                return
        if a < current.key:
                self._range(a, b, current.left, a_list)
        if a <= current.key <= b:
                a_list.append(current.key)
        if current.key < b:
                self._range(a, b, current.right, a_list)
```

Exercise 6

Given a BST with numeric values and a number $k > 0$, write down an algorithm that returns the k-largest element in the BST. For $k = 1$ it returns the largest element, for $k = 2$ the second largest, and so on. The idea is to do this without visiting all elements, if possible.

Solution

```
def k_largest(self, k):
        a_list = []
        self._k_largest(self.root, k, a_list)
        return a_list[-1]

def _k_largest(self, current, k, a_list):
        if current is not None:
                ans = self._k_largest(current.right, k, a_list)
                if ans is not None:
                        return ans
                a_list.append(current.key)
                if len(a_list) == k:
                        return a_list
                ans = self._k_largest(current.left, k, a_list)
                if ans is not None:
                        return ans
```