Python Intermediate Programming

Unit 3:Python Data Structures

CHAPTER 8: TREE PART 1

DR. ERIC CHOU

IEEE SENIOR MEMBER

Objectives

- In this lecture we upgrade our discussion of self-referential structures from linked lists to (binary) trees, by creating TN: a class that includes a value and two references to other objects from the TN class (or None). What seems like a trivial extension turns out to be profound: like going from a 1-dimensional world to a 2-dimensional world. There are entire books written about trees (in both computer science and mathematics), but no books written solely about linked lists.

Overview

LECTURE 1

Overview

- Over the next two lectures we will examine a few applications for trees. We will discuss binary ordered trees (search trees) and structure trees (expression trees) and discuss various recursive functions that operate on them. Both use the same definition of the TN (tree node) class shown below

```
class TN: # A binary tree: each TN has two children
    def __init__(self : "TN",  \
                 value : object, left : "TN" = None,   \
                 right : "TN" = None):
        self.value = value
        self.left  = left
        self.right = right
```

Overview

- We write "TN" in the annotations above, because when defining TN we cannot use TN for an annotation (because it hasn't been completely defined yet).
- We will discuss many operations on trees below, written as functions. We can also define methods that implement these operations as methods in the TN class.

Binary Search Trees

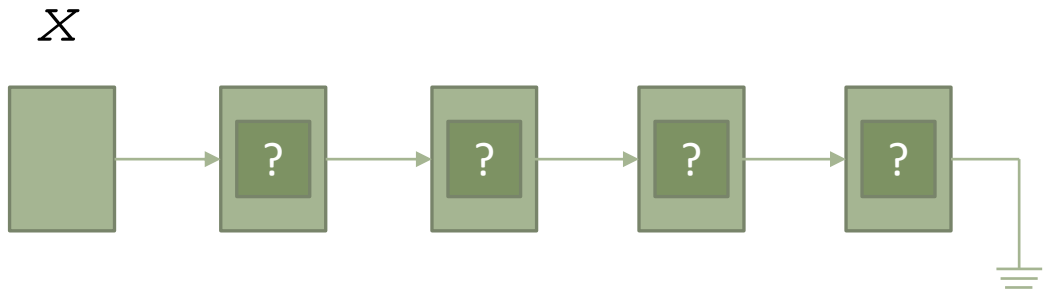
LECTURE 2

Binary Search Trees

- Binary trees have structure and order property. Its structure property dictates that every parent node has 0, 1, or 2 children nodes (called left and right; each is another binary tree or None). We draw binary trees with their roots on the top, their left and right children below, and their leaves at the bottom (a leaf is a node with 0/no children: self.left and self.right are both None; an internal node has at least one non-None child). A binary search tree (one special kind of binary tree) also has an order property: it dictates that all values in the left subtree of any node are less than that node, and all values in the right subtree of any node is greater than that node. Typically, binary search trees store unique values (and we will assume so in this lecture).

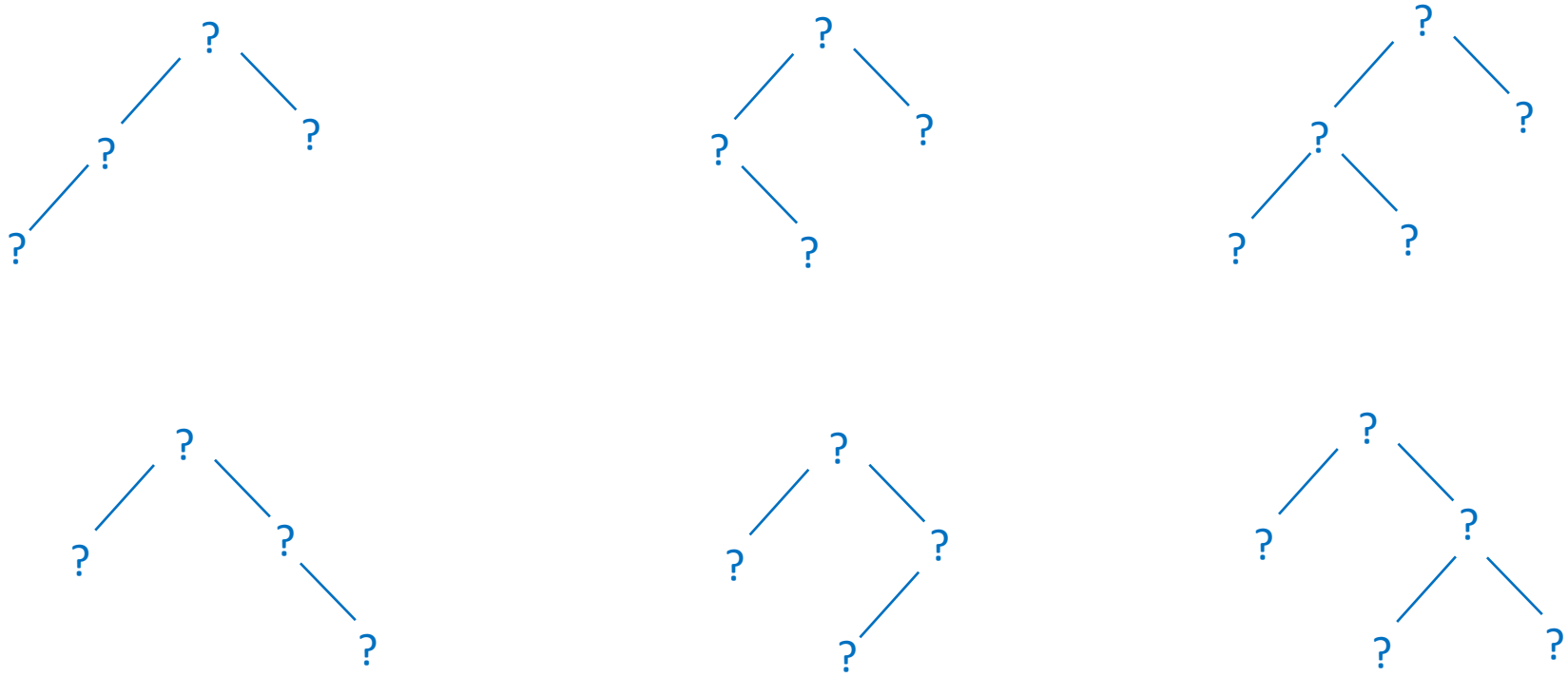
Binary Search Trees

- Structurally, binary trees are much more interesting than linked lists: structurally (ignoring values) there is only one linked list of length 4:

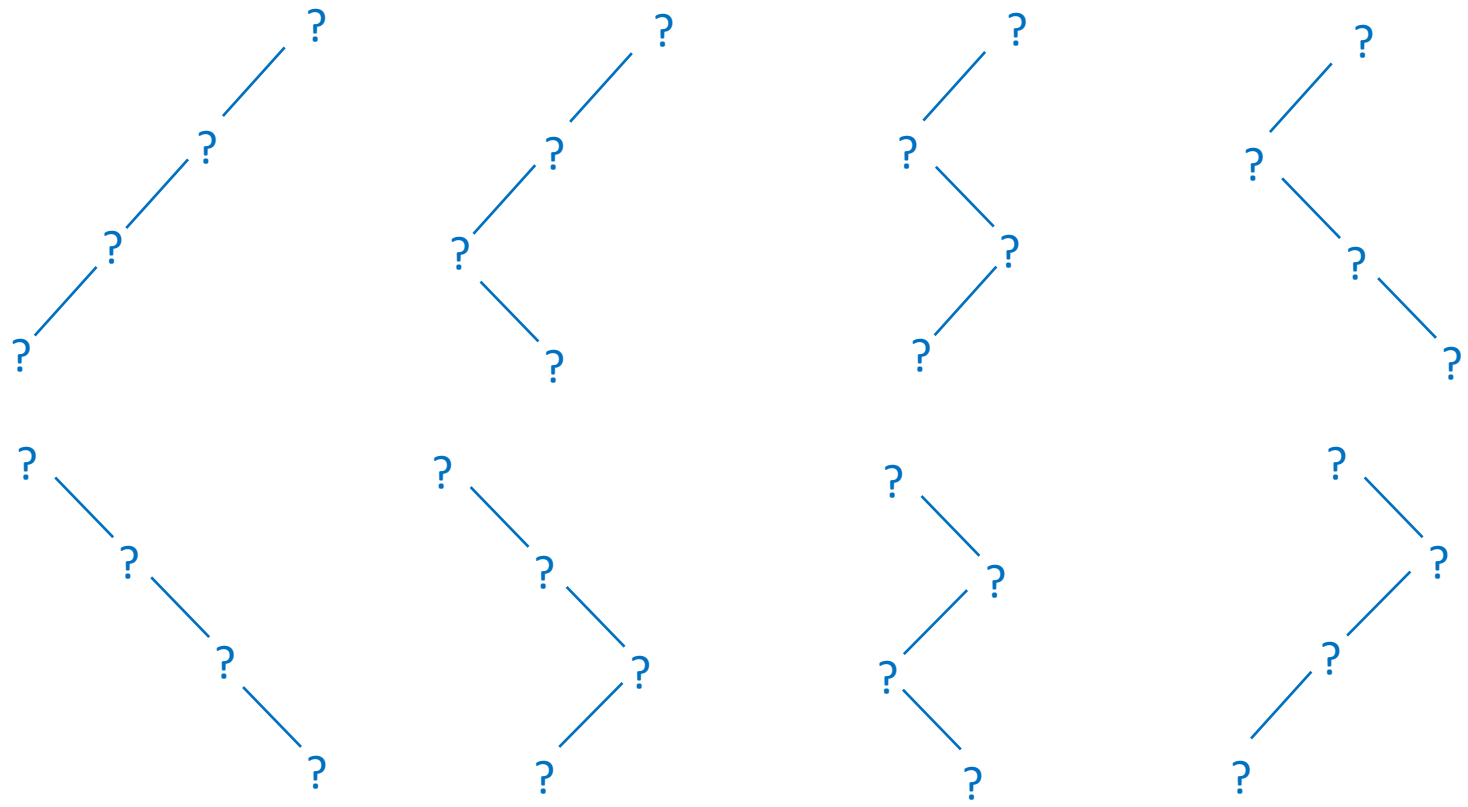


- But there are 14 different binary trees with four nodes. Here is a listing of all 14. They are arranged in two groups, such that in each group a tree and its mirror image are above/below each other.

Binary Search Trees

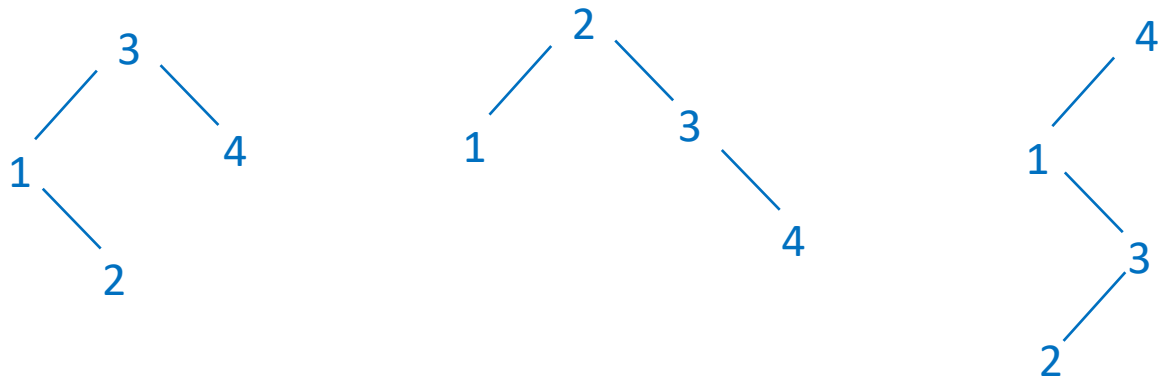


Binary Search Trees



Binary Search Trees

- Note that the shape of a binary search trees is not uniquely determined by the values that it contains. For example, a binary search tree with the values 1, 2, 3, and 4 can be represented by



Binary Search Trees

or ... any of the 14 structures above, with the right selection of node values.

- Note that for EVERY NODE in the binary search trees above (not just the ROOT), the parent is > all values in its left subtree and < all values in its right subtree.
- Later, when we study the add function, we will learn that the shape of a binary search tree is determined not just by the values that it contains, but also by the order in which these values were added to the binary tree.

Metrics

LECTURE 3

Metrics

- There is just one standard metric for linked lists: length. For trees there are two standard metrics: size and height. Size counts the number of nodes in a tree (therefore it is similar to length for linked lists). It is easy to compute size recursively, using a function similar to a recursive computation of the length of a list (but with two recursive call -one for computing the size each subtree- instead of one -for the nodes following in the list).

```
def size(atree):
    if atree == None:
        return 0
    else:
        return 1 + size(atree.left) + size(atree.right)
```

Metrics

- There is no simple way to compute size with a loop: for every node in the tree we must visit both its left and right subtrees, so every time that we go left we must also save the right for future reference so that we can go there too; we can write this function iteratively by using an extra list of nodes, but the code is not simple to write nor easy to understand.

```
def size_i(atree)
    nodes = []
    size = 0
    nodes.append(atree)
    while len(nodes) > 0:
        next = nodes.pop(0)
        if next != None:
            size += 1
            nodes.append(next.left)
            nodes.append(next.right)
    return size
```

Metrics

- The second metric for trees is height. The standard definition of the height of a node is a bit strange: it is the number of steps needed to get from the node to the deepest leaf in either of the node's subtrees. So the height of a leaf (the base case) is 0 and the height of a tree is the height of its root. We can directly translate this definition into the following code. Again there are (at most) two recursive calls, in the case of a node with two non-None children.

```
def height(atree)
    if atree.left == None and atree.right == None:    # leaf check as base case
        return 0
    elif atree.right == None:                         # only a left subtree
        return 1 + height(atree.left)                 #   recur only to left
    elif atree.left == None:                          # only a right subtree
        return 1 + height(atree.right)                #   recur only to right
    else                                              # both a left/right subtree
        return 1 + max(height(atree.left),height(atree.right)) # recur on both
```

Metrics

- This function deals with all the necessary cases: a leaf node, an internal node with only a left (or only a right) subtree, and an internal node with both left and right subtrees. This function does not work on empty trees, which have no directly defined height, given the previous definition.

Metrics

- But, this code is much more complicated than the code for computing size. The complexity results from using a leaf node as the base case. Let us simplify this code by using an empty tree as a base case, even though it makes no sense for the definition of the height of a node:
- The number of steps needed to get from the node to the deepest leaf in either of the node's subtrees.
- In an empty tree, we have no node to start at and no leaf to reach.

Metrics

- With this new definition, we will "arbitrarily" define the height of an empty tree to be -1. This might seem like a very strange approach, but it seems reasonable too: an empty tree should have a height that is one less than a leaf node (whose height is 0). By using this definition (and no others), we can simplify the height function dramatically (as well as defining it for all possible trees, even empty ones).

```
def height(atree):
    if atree == None:
        return -1
    else:
        return 1 + max(height(atree.left),height(atree.right))
```

Metrics

- Mathematicians generalize definitions such as this one all the time. For any value a, a**0 is defined as 1. There are many ways to justify this definition (some quite complicated, using limits and calculus); the simplest way is to note the algebraic law a**x * a**y = a **(x+y). By this law (a quite useful one to have) a**0 * a**x = a**(0+x) = a**x; which means that a**0 must be equal to 1 for this identity to hold.

Metrics

- If we couldn't guess that -1 was the correct answer, we could deduce it. If we started by writing

```
def height(atree):
    if atree == None:
        return empty-height
        # actual value of empty-height to be determined
    else:
        return 1+ max(height(atree.left),height(atree.right))
```

Metrics

and looked at height called on a leaf node (which we know must compute a height of 0), we would have

```
# height(None) because it is a leaf
0 = 1 + max(height(None),height(None))
# height(None) returns empty-height
0 = 1 + max(empty-height,empty-height)
# max(x,x) = x for all x
0 = 1 + empty-height
# subtract 1 from each side
-1 = empty-height
```

Metrics

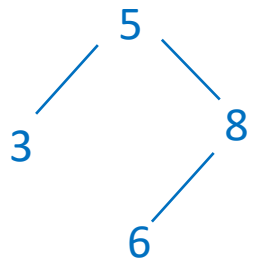
- The second line comes from computing the height of each base case; the third comes from simplifying that max(x,x) = x (the maximum of a value and itself is that value); the fourth line comes from subtracting 1 from each side of the equality. So, we have deduced (from the recursive call) what the base case (None) should return -1.

Converting between Binary Tress and Lists

LECTURE 4

Converting between Binary Tress and Lists

- Next, we will look at functions that convert between trees and lists, showing that there is a standard way to represent a tree as a nested list of values. We represent every TN as a 3-list containing (in order) the value, left, and right subtrees (each subtree is itself a 3-list). So, we represent the tree



- by the list [5, [3, None, None], [8, [6, None, None], None]]. Note that each list in this data structure always has exactly 3 values (empty subtrees will be None). We could also put the value in the middle of the 3-list, which would result in [[None, 3, None], 5, [[None, 6, None], 8, None]] for the tree above.

- These lists can be deeply nested for tall trees.

Converting between Binary Tress and Lists

- There are simple recursive functions to translate a tree argument returning a list, and a list argument returning a tree. Again, each uses two recursive calls

```
def list_to_tree(alist):
    if alist == None:
        return None
    else:
        return TN(alist[0],list_to_tree(alist[1]), \
                  list_to_tree(alist[2]) )
```

- Each recursive call on a non-empty list builds a TN with a value (alist[0]), and then produces subtrees from the next two values in the 3-list; eventually None will be reached as base cases.

Converting between Binary Tress and Lists

- We can also easily translate from a tree (TN) to a list.

```
def tree_to_list(atree):
    if atree == None:
        return None
    else:
        return [atree.value,
                tree_to_list(atree.left),
                tree_to_list(atree.right)]
```

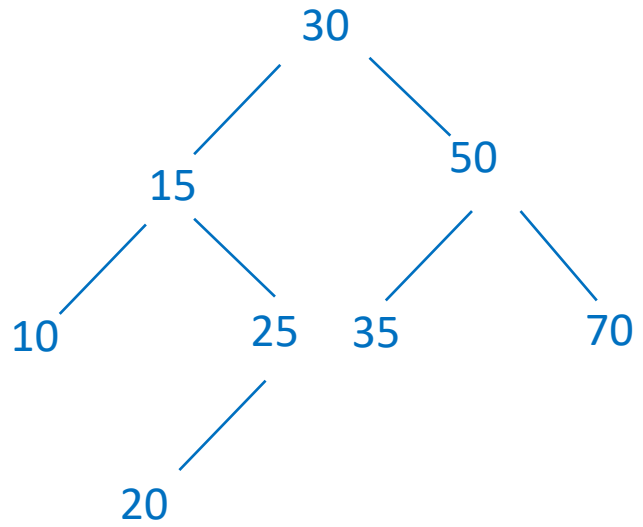
- Each recursive call on a non-empty tree builds a 3-list of the value, followed by the list equivalent of the left and right subtrees; eventually None will be reached as base cases.

Printing a Binary Tree

LECTURE 5

Printing a Binary Tree

- The following function prints a tree rotated 90 degree counter-clockwise. So the tree we would show as



prints as follows. Notice where the root (30) appears, and where the roots of its left (15) and right (50) subtrees appear, and the left/right roots of those subtrees, etc.

Printing a Binary Tree

```
. . . . 70
. . 50
. . . . 35
30
. . . . 25
. . . . . . 20
. . 15
. . . . 10
```

Printing a Binary Tree

- This function declares print_tree_1, as a local helper function that does all the recursive work (using the indent_char/indent_delta parameters), and then calls print_tree_1 with an initial identation of 0. The helper function either does nothing (for printing an empty tree), or prints all values in its right subtree (first, with more indentation), its own value, and then all values in its left subtree (with more indentation).

```
def print_tree(atree,indent_char =' ',indent_delta=2):
    def print_tree_1(indent,atree):
        if atree == None:
            return None     # print nothing
        else:
            print_tree_1(indent+indent_delta, atree.right)
            print(indent*indent_char+str(atree.value))
            print_tree_1(indent+indent_delta, atree.left)
    print_tree_1(0,atree)
```

Printing a Binary Tree

- At this point, we have dealt with the structure of trees, but not their values. In a binary search trees, we can use its extra order property to search for, add a value, and remove a value efficiently: think of a tree representing a set of values (each value in a set is unique; that mirrors our intent of having unique values in binary trees).

Searching for a value in a Binary Search Tree

LECTURE 6

Searching for a value in a Binary Search Tree

- We can use the following iterative function to search for a value; unlike the other functions written above, this one goes only one way (left or right) for each tree node. We know that if the value we are searching for is less than a node's value, by the order property of a binary search tree it must be in the left subtree; if the value we are searching for is greater than a node's value, it must be in the right subtree.

```
def search_i(atree,value):
    while atree != None and atree.value != value:
        if value < atree.value
            atree = atree.left
        else:
            atree = atree.right
    return atree  # either None or the TN storing value
```

Searching for a value in a Binary Search Tree

- Note that the if statement is selecting which value (atree.left or atree.right) to store in atree, so we can simplify this if statement using a conditional expression.

```
def search_i(atree,value):
    while atree != None and atree.value != value:
        atree = (atree.left if value <          \
                 atree.value else atree.right)
    return atree  # either None or the TN storing value
```

Searching for a value in a Binary Search Tree

- We can also write this function recursively.

```
def search_r(atree,value):
    if atree == None:
        return None
    else:
        if value == atree.value:
            return atree
        elif value < atree.value:
            return search_r(atree.left,value)
        else: # value > atree.value
              #true by law of trichotomy: ==, <, or >
            return search_r(atree.right,value)
```

Searching for a value in a Binary Search Tree

- We can combine the base check and equality check, and use a conditional expression to shorten this function to the following

```
def search(atree,value):
    if atree == None or atree.value == value
        return atree
    else:
        return search( (atree.left if value < atree.value        \
                                   else atree.right), value)
```

- In the function above, the "base" case is an empty tree or the node storing the value; the same recursive call is executed for subtrees, with the first "smaller" tree (having fewer nodes) being either atree.left or atree.right.
- Because this is a tail-recursive function, we expect to be able to write it iteratively (as we did above).

Adding/Removing a value to/from a Binary Search Tree

LECTURE 7

Adding/Removing a value to/from a Binary Search Tree

- Now, here is a similar (to the top) function to add a value to a tree. We call it like: atree = add(atree,value) -similarly to how we added a value to a list.

```
def add(atree,value):
    if atree == None:
        return TN(value)
    if value < atree.value:
        atree.left = add(atree.left,value)
        return atree
    elif value > atree.value:
        atree.right = add(atree.right,value)
        return atree
    else: # value == atree.value
          # true by law of trichotomy: ==, <, or >
        return atree  # already in tree; do not change the tree
```

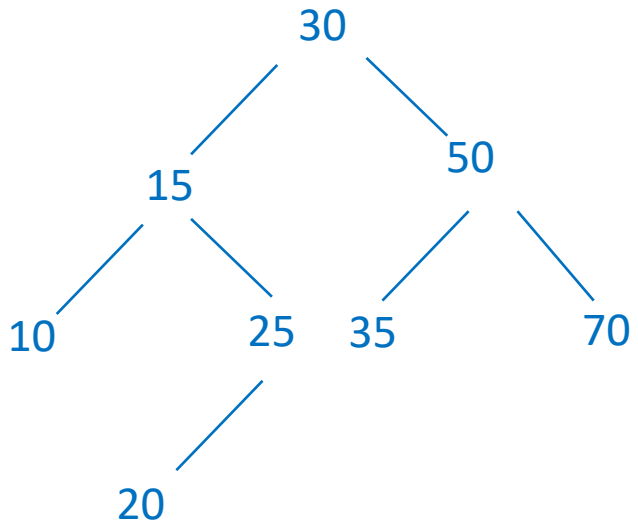
Adding/Removing a value to/from a Binary Search Tree

In all cases, this function returns a tree to which a TN with value has been added as a subtree (returning all the values in the original tree including the new node/value). By the 3 proof rules.

1. The code detects the base case (an empty tree) and return a tree containing only a node storing value (all the nodes in the original tree -there are none- including a node storing value).
2. Each recursive call (there are two) is on a left or right subtree (which is smaller than the entire tree, at least by one node, probably by many more if the other side contains some nodes).
3. Assume calling add returns a new tree containing all the nodes in its smaller argument tree, including a node containing value. When the value is equal to atree.value, it returns just atree (which already contains value, not duplicating that value). When the value is less/greater than atree.value, it calls add recursively, which returns the left/right tree with value included, and stores it in atree.left/atree.right; finally it returns atree, which is tree containing value (either in left/right subtree of atree).

Adding/Removing a value to/from a Binary Search Tree

- Recall that the structure of a tree is not determined solely by the values it contains. As we saw above, there are many legal binary search trees storing the same values. The structure is determined by the order those values are added to the tree. Adding values in increasing order, decreasing order, at random, will all produce different shaped trees.
- I will defer showing the remove function, but I will describe it here and you should use this description to practice deleting values from trees (shown pictorially). Use the following simple tree for a first example



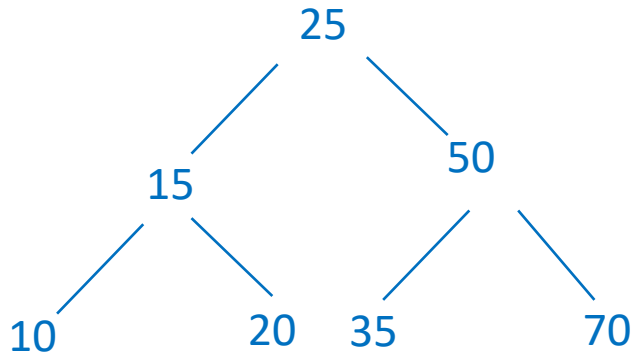
Adding/Removing a value to/from a Binary Search Tree

Here are the rules:

1) To remove a value in a leaf, make its parent refer to None
2) To remove a value in a node with one child, make its parent refer to its child (this works whether the node is a left/right child of its parent, and whether its child is a left/right child)
3) To remove a value in a node with 2 children:
   a) Find the biggest node less than it (or smallest node greater than it) that node must have either 0 or 1 children (can you explain why?)
   b) Remove that node by rule 1 or 2
   c) Take its value and put it as the value of the node being removed. So, the node being removed isn't really removed (another one is): but, its value is replaced by another value, so the value is removed

Adding/Removing a value to/from a Binary Search Tree

- The first two rules are very simple. Here is an example of applying the third. If we remove the value at the root, 30, we would (a) find the node 25, (b) remove the value here by making 15's right refer to 20, (c) move the value 25 to the node that contains 30. Note the order property is preserved: all values to the left of the node that used to store 30 are less than what it now stores, 25 (25 was the biggest of the nodes < 30); all values to the right of the node that used to store 30 are greater than what it now stores, 25 (25 is < 30, so nodes > 30 are > 25).



Adding/Removing a value to/from a Binary Search Tree

- The binarysearchtree module contains simple recursive functions for copying a tree and determining whether two trees are equal (not only store the same values overall, but store trees that have these values in the same shape). Examine those functions, which appear below (or better yet, try to write them yourself first).

```
def copy(atree):
    if atree == None:
        return None
    else:
        return TN(atree.value, copy(atree.left), copy(atree.right))
def equal(t1,t2):
    if t1 == None or t2 == None:
        return t1 == None and t2 == None
    else:
        return t1.value == t2.value and
               equal(t1.left,t2.left) and
               equal(t1.right,t2.right)
```

Adding/Removing a value to/from a Binary Search Tree

- Note that for the short-circut "and" operator in equal, if the values in any node are not equal, the value False is returned immediately, without making the recursive calls to equal.
- In that module the generator_in_order generator yields all the values (from lowest to highest) in the tree it is called on. In the next lecture we will study traversal orders more generally, discussing pre-order, in-order, post-order, and breadth-first order.

Adding/Removing a value to/from a Binary Search Tree

- We can use binary search trees easily to reprsent a dictionary: each TN would store a value that is 2-tuple, a key-value pair. The keys in a dictionary are known to be unique. When processing a tree, Python will always compare/process the first value in the 2-tuple (the key). In a binary search tree representation of a dictionary, all keys must be comparable; in a Python dict, we can have keys that aren't comparable: one key could be an int and another a str. So, Python dictionaries are NOT represented by binary search trees, but by something even faster: hash tables. ICS-46 covers runtime performance (efficiency) of lists, trees, and hash tables (which is how Python stores both sets and dictionaries; hash tables are covered in a later ICS-33 lecture note).

Adding/Removing a value to/from a Binary Search Tree

- A well-balanced binary search tree (all nodes having about an equal number of children in its left and right subtrees) can be searched much faster than a list or linked list. The amount of time it takes to search any binary search tree is bounded by its height: the number of comparisons in needs to go downward in the tree until it reaches the value it is searching for (or goes beyond a leaf, meaning that the value is not in the tree).
- The height of an N-node tree must be at least Log2 N (log base 2 of the number of nodes in the tree). The typical height, when values are added randomly, is 2-3 times that. In a linked list (or pathological binary search tree: a very deep skinny one) the number of comparisons is N. Log2 N is generally a much smaller number than N: Log2 1,000 is about 10; Log2 1,000,000 is about 20; and Log2 10^9 (a billion) is about 30.

Adding/Removing a value to/from a Binary Search Tree

- Try the following experiment, which prints the height of a tree with 1,000 values, added in a random order.

```
values = [i for i in range(1000)]
random.shuffle(values)
print(height(add_all(None,values)))
```

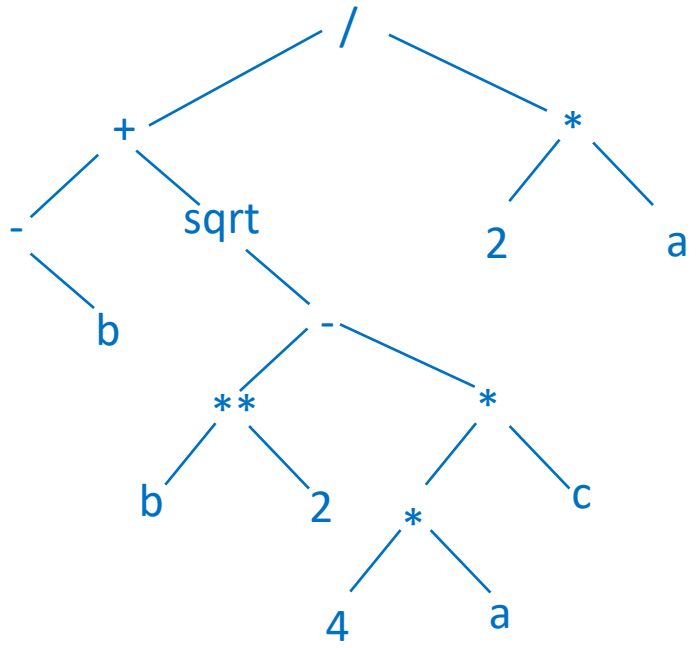
- Log2 1,000 is about 10, so the typical height of such a tree is about 20-30, which means it takes 20-30 comparisons to find a value: much better than the average of about 500 if the values are in an unordered list or linked-list. Also, see the random_height function in the binarysearchtree.py module.
- Again, in ICS-46 we will look at tree processing in more depth :).

Expression Trees

LECTURE 8

Expression Trees

- We can also use binary trees to represent expressions. In these trees, leaf nodes represent values (either literals or names bound to values), and the internal nodes represent binary operators or unary operators or unary functions (whose operands will be in the right subtree). For example, the expression (-b + sqrt(b**2 - 4*a*c))/(2*a) would be represented by the expression tree.

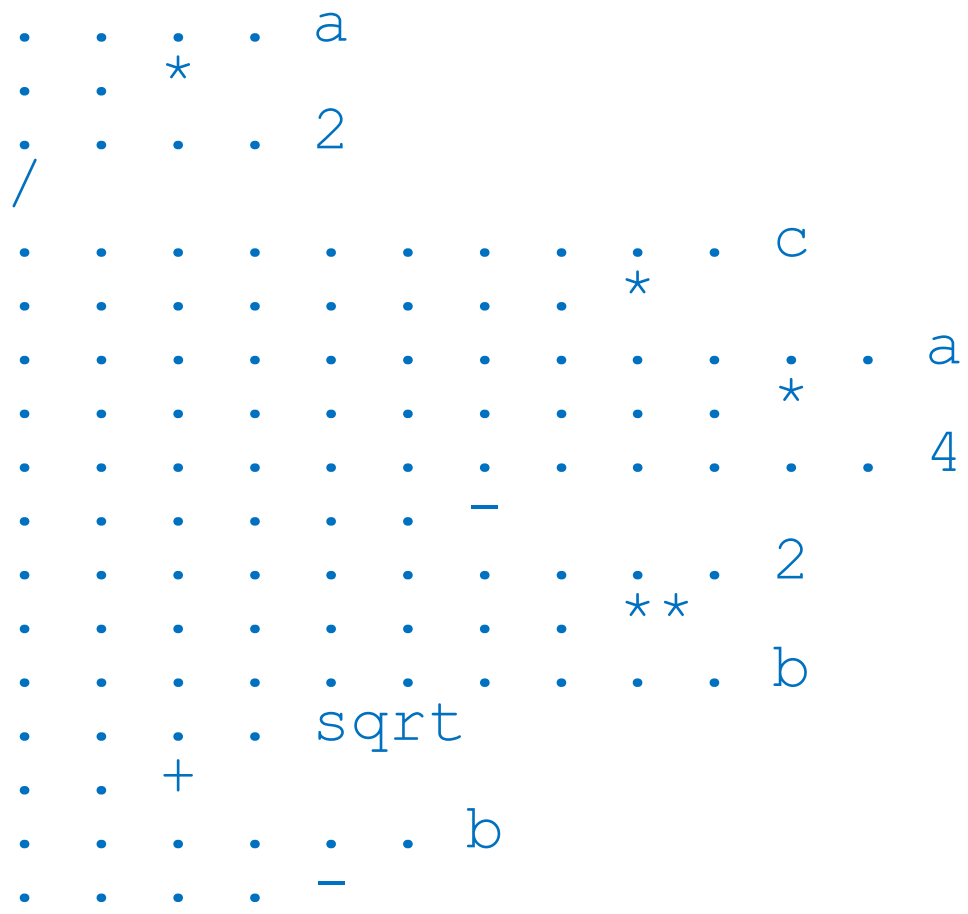


Expression Trees

- Here I wrote '/' for the divide operator, since / means a left subtree.
- Note that the structure of the tree determines how the subexpressions are computed. There is no need for operator precedence rules or parentheses: the structure of the tree embodies these rules.
- There is an algorithm that people can follow to construct such a tree: find the last operator or function call the computer would evaluate and put that at the root of the tree; now do the same for its one/two subtrees that are sub-expressions, and keep repeating finding the root of these until there are no more operators or functions (names and literals stand for themselves).

Expression Trees

- In the expression above, the division between the numerator and denominator is evaluated last: on the left side the addition is evaluated last; on the right side there is only the multiplications, so that is done last. Continue this process. If we call print_tree on this tree, it would print



Expression Trees

- Once we have such a tree, we can perform many operations on it. The first and most important is evaluating the tree. We can do this recursively (evaluating subexpressions) by
  1. evaluating leaves as themselves
  2. evaluating either unary operators on their evaluated operand or unary functions on their evaluated argument
  3. evaluating binary operators on their evaluated arguments

Expression Trees

- The code for this method follows this outline

```
def evaluate(etree):
    #name/literal
    if etree.left == None and etree.right == None:
        return eval(str(etree.value))
    #unary operator/function cal
    elif etree.left == None:
        if etree.value in {'+','-'}:
            #unary operator
            return eval(etree.value + str(evaluate(etree.right)))
        else:
            #function call: assume legal name
            return eval(etree.value+'('+str(evaluate(etree.right))+')')
    else:
        #binary operator: assume etree.value in {'+','-','*','/','//','**'}

        return eval(str(evaluate(etree.left)) + etree.value
                    + str(evaluate(etree.right)))
```

- If we set a=1, b=2, c=1, the calcuated value is -1.

Expression Trees

- We can translate this tree into infix (but overparenthesized) and postfix form: in the postfix form, each operator is proceeded by its two operands: "a + 1“ (infix form) translates to "a 1 +" (postfix form). Using postfix notation (also called Polish notation because it was invented by Polish logicians right before World War II), we can write expressions unambiguously without any parentheses or knowledge of operator precedence! "(a + b) * c" translates to "a b + c *“ and "a + b * c" translates to "a b c * +".

Expression Trees

- Here are the functions to perform these translations, and their results.

```
def infix(etree):
    if etree.left == None and etree.right == None:
        return '('+str(etree.value)+')'
    elif etree.left == None:
        return '('+etree.value+str(infix(etree.right))+')'
    else:
        return '('+str(infix(etree.left))       \
               + etree.value+str(infix(etree.right))+')'
```

which produces:

```
(((-(b))+(sqrt(((b)**(2))-(((4)*(a))*(c)))))/((2)*(a)))
```

which is correctly but over parenthesized

Expression Trees

```
def postfix(etree):
    if etree.left == None and etree.right == None:
        return str(etree.value)
    elif etree.left == None:
        return str(postfix(etree.right)) + ' ' + etree.value
    else:
        return str(postfix(etree.left))     \
               + ' ' + str(postfix(etree.right))     \
               + ' ' + etree.value
```

which produces:

```
b - b 2 ** 4 a * c * - sqrt + 2 a * /
```

Expression Trees

- If you have never seen Polish notation this is difficult to read, but if you have studied this notation, it is easy. To understand which operators apply to which data, start on the left and circle each operand: when you get to an operator circle it and the number of operands it takes (which all come before it). Look at smalller examples: 1+2*3 is 123*+ while (1+2)*3 is 12+3*. The operands in polish notation appear in the same order as regular notation, but the operators appear in different spots based on operator precedence and parentheses.
- Finally, I have defined a parse_infix function that takes a string argument and produces a tree representing the string. It is limited in the following ways: all tokens must be separated by spaces; it assumes all operators are binary, and that all operators are left-associative (which ** is not). So, it does a bit of what Python does when it processes expressions written in Python, but doesn't do everything correctly.