

# CS313E: Elements of Software Design

## Trees

Dr. Bill Young  
Department of Computer Sciences  
University of Texas at Austin

Last updated: November 23, 2011 at 06:35

# Abstract Data Types

Recall that the basic idea of object oriented programming (OOP) and also of abstract data types is to view the world as a collection of objects.

Each object has:

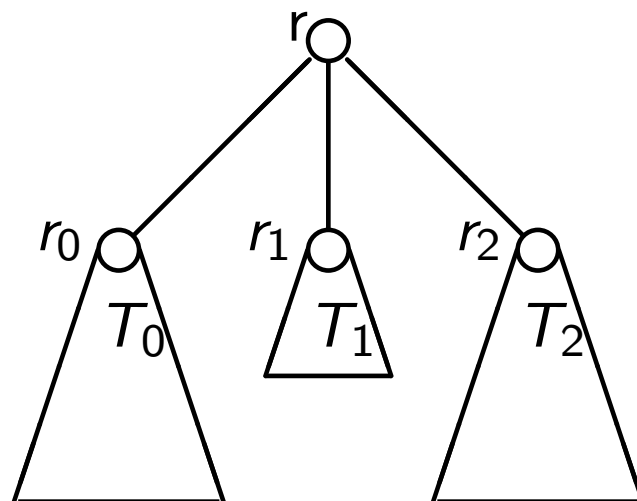
- some *data* that it maintains characterizing it's current state;
- a set of actions (*methods*) that it can perform.

The programmer interacts with these objects by invoking their methods, which may:

- update the state of the object,
- query the object about its current state,
- compute some function of the state and externally provided values,
- some combination of these.

So far we have encountered at least the following Abstract Data Types:

- Stacks
- Queues (including Bounded Queues and Priority Queues)
- Linked Lists



A (*rooted*) *tree* is composed of *nodes* (vertices) and *edges* in which one of the nodes is distinguished as the *root*. An edge is an ordered pair  $\langle u, v \rangle$  of nodes.

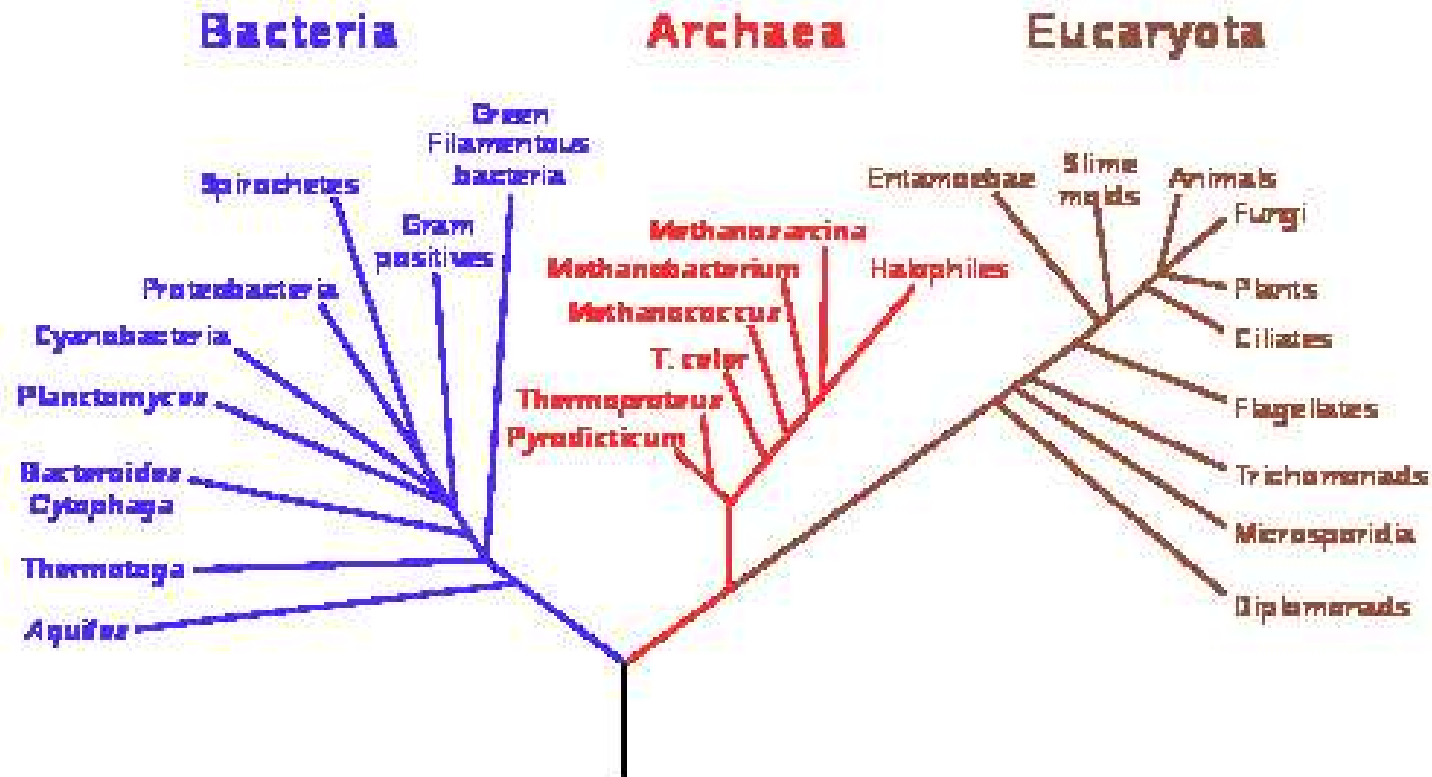
We consider a single node with no edges to be a tree. That node is also the root of the tree.

A tree must be *acyclic*, meaning that there are no loops in the tree structure.

# A Sample Tree

We often associate an item or a label with each node of a tree, so trees can naturally represent hierarchical relationships.

## Phylogenetic Tree of Life



# Why Trees?

**Example:** the file structure on a Linux/Unix system is naturally represented as a tree.

Consider the directories: `/`, `/usr`, `/u`, `/etc`, `/usr/bin`, `/usr/local`, `/usr/include`, `/usr/local/bin`, `/usr/local/include`, `/u/mitra`, `/u/byoung`.

What tree represents this hierarchy?

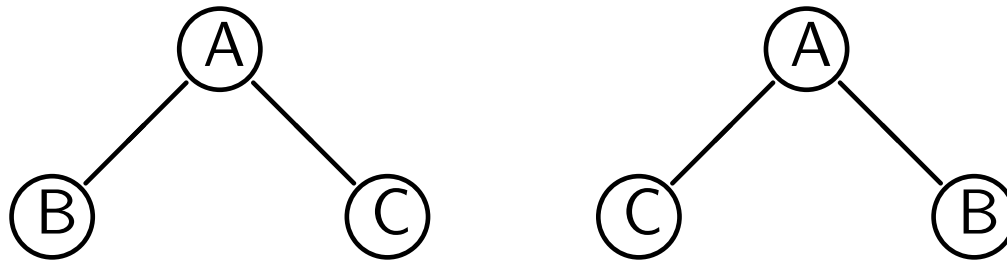
A node  $r$  may be the *parent* of  $r_0, r_1, \dots, r_{k-1}$ , which are *children* of  $r$  and the *siblings* of each other.

- The root of the tree is the only node with no parents.
- For any two nodes  $u$  and  $v$ , if  $u$  is on the unique path from  $r$  to  $v$ , then  $u$  is an *ancestor* of  $v$  and  $v$  is a *descendent* of  $u$ .
- A *leaf* is a node with no children.
- A non-leaf node is an *internal node*.
- The *subtree rooted at  $u$*  is the tree consisting of descendents of  $u$ , rooted at  $u$ .

- Any tree has one more node than edges.
- The number of children of a node  $u$  is called the *degree* of  $u$ .
- The length of the path from the root  $r$  to a node  $u$  is called the *depth* of  $u$ . The root has depth 0.

# Ordered Trees

An *ordered tree* is a tree in which we care about the order of the children of each node. That is, if a node has  $k$  children, then there is a first child, a second child,  $\dots$ , and a  $k$ th child. We often consider the children as being ordered from left to right, so the first child is the leftmost, and the  $k$ th is the rightmost.



These are different ordered trees, but the same trees if considered as unordered.

# Expression Trees

Ordered trees are also good for representing arithmetic expressions and grammatical expressions.

**Example:** What do you think are the expression trees for the following expressions?

$$2 * 3 + 4$$

$$2 * (3 + 4)$$

$$(5 + 8) * (7 - 2 * 3 + 9)$$

How do these trees relate to our infix, postfix, and prefix representations of arithmetic expressions?

# Binary Trees

An *binary tree* is (usually) an ordered tree in which each node has at most 2 children. Moreover, each child of a node is distinguished as being either a *right child* or a *left child*, even when the node has only one child.



These are different binary trees, but the same trees if considered simply as ordered trees or unordered trees.

Notice that a binary tree is (usually) ordered, but that doesn't imply that the values stored in the tree have a natural ordering relation.

For convenience, we extend the notion of a binary tree to include the *empty binary tree*, which contains no nodes.

The idea of a binary tree is naturally extended to *k-ary trees*. In a *k-ary tree*, each node has degree at most *k*.

A *k-ary tree* has *arity k*. Thus, a binary tree has arity 2.

# Binary Tree ADT Interface

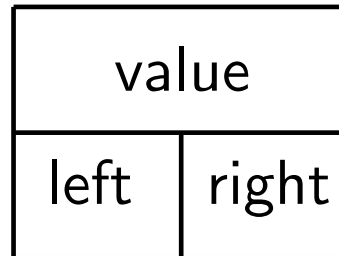
What are the operations you'd like to perform on a Binary Tree?

- is the Tree empty
- get value at the root
- get the left subtree
- get the right subtree
- visit all nodes in the tree
- others?

# Binary Tree Implementation

In the most natural implementation, each node in the binary tree is a 3-tuple, consisting of:

- value
- left child
- right child



# BinaryTreeNode

```
class BinaryTreeNode:
    """A BinaryTreeNode has three fields:
    - a value, which can be arbitrary
    - a left pointer to another BinaryTreeNode
    - a right pointer to another BinaryTreeNode"""

    def __init__(self, value, left=None, right=None):
        self._value = value
        self._left = left
        self._right = right

    def getValue(self):
        return self._value
    def getLeft(self):
        return self._left
    def getRight(self):
        return self._right

    # Should probably also add mutators: setValue,
    # setLeft and setRight.

    def isLeafNode(self):
        return not self._left and not self._right
```

# Binary Tree

Notice that a `BinaryTree` is really just a pointer (possibly `None`) to a `BinaryTreeNode`. It's a “wrapper” to turn a node into a tree.

```
class BinaryTree:
    """A BinaryTree really is just a pointer to a BinaryTreeNode."""

    def __init__(self, root=None):
        """To create a tree, pass a pointer to a BinaryTreeNode;
        defaults to None."""
        self._root = root

    def isEmpty(self):
        return self._root == None

    def getRoot(self):
        return self._root
```

# Watch the Types

Notice that `BinaryTreeNode` and `BinaryTree` *are two different classes*, so don't confuse them.

Much of the time when you're constructing or visiting a `BinaryTree`, you're really manipulating `BinaryTreeNode` items.

# Example Code

Below is some code to create and query a binary tree:

```
>>> from Trees import *
>>> n1 = BinaryTreeNode(2)
>>> n3 = BinaryTreeNode(4)
>>> n2 = BinaryTreeNode(3, n1, n3)
>>> tree = BinaryTree(n2)
>>> tree.getRoot()
<Trees.BinaryTreeNode object at 0xb75da5ec>
>>> tree.getRoot().getValue()
3
>>> tree.getRoot().getLeft().getValue()
2
>>> tree.getRoot().getRight().getValue()
4
```

# An Application

Convert a postfix expression into a binary expression tree. (You actually build a “tree” of nodes and then put the BinaryTree “wrapper” on it at the end.

- ① Begin with an empty stack.
- ② Read token from input:
  - ① If token is a number, convert to a node and push onto the stack.
  - ② If operator of arity  $k$ , pop  $k$  trees from the stack, create a node with  $k$  children and push onto the stack.
  - ③ If input is empty, answer is on top of stack. Convert it to a BinaryTree and return it.

Example: convert  $2\ 3\ *\ 4\ +\ 2\ *$  to a binary expression tree.

# Postfix to Binary Tree

```
def postfixToTree(postfixExpr):
    stack = Stack()
    tokens = postfixExpr.split()
    for token in tokens:
        # Token is a number
        if isInteger( token ): # need to write this
            node = BinaryTreeNode( token )
            stack.push( node )
        elif isOperator( token ): # need to write this
            if len( stack ) < arity( token ): # need to write
                print( "Ill-formed expression" )
                return

            # assumes operator is binary, change for general case
            arg2 = stack.pop(); arg1 = stack.pop()
            node = BinaryTreeNode( token, arg1, arg2 )
            stack.push( node )
        else:
            print( "Token not recognized: ", token )
            return
    if len( stack ) > 1:
        print( "Ill-formed expression"); return
    return BinaryTree( stack.pop() )
```

# Testing our Code

```
>>> from Trees import *
>>> expr = "2 3 * 4 + 2 * 5 6 + *"
>>> tree = postfixToTree( expr )
tree.getRoot()
<Trees.BinaryTreeNode object at 0xb76d902c>
>>> tree.getRoot().getValue()
'*'
>>> tree.getRoot().getLeft().getValue()
'*'
>>> tree.getRoot().getRight().getValue()
'+'
>>> type(tree.getRoot())
<class 'Trees.BinaryTreeNode'>
>>> type(tree)
<class 'Trees.BinaryTree'>
>>> root = tree.getRoot()
>>> root.getRight().getValue()
'+'
```

To *traverse* a tree means to visit each node of a tree in a particular order. There are several common traversals on an ordered tree.

- Preorder traversal
- Inorder traversal
- Postorder traversal
- Level-order traversal

How many different types of traversals are possible?

# Preorder Traversal

In a *preorder traversal* we visit each node *before* visiting all of the node's descendants.

- 1 visit the node
- 2 traverse the left subtree
- 3 traverse the right subtree

Given an expression tree for  $2 + 3 * 4$ , what is the result of a preorder traversal?

How would you change this algorithm for a  $k$ -ary tree?

# Preorder Traversal Code

This is in the BinaryTree class:

```
# Notice that this is a class method, not an instance method.
def preorderAux(node):
    """Do a preorder traversal from the node."""
    if node == None:
        return
    else:
        print (node.getValue(), end=" ")
        BinaryTree.preorderAux(node.getLeft())
        BinaryTree.preorderAux(node.getRight())

def preorder(self):
    if self.isEmpty():
        print("The tree is empty.")
    else:
        BinaryTree.preorderAux(self._root)
```

# Inorder Traversal

In an *inorder traversal* we traverse a node's left subtree, visit the node, and then traverse the node's right subtree.

- 1 traverse the left subtree
- 2 visit the node
- 3 traverse the right subtree

Given an expression tree for  $2 + 3 * 4$ , what is the result of a inorder traversal? Can you put in parentheses as you traverse?

How would you change this algorithm for a  $k$ -ary tree?

# Postorder Traversal

In a *postorder traversal* we traverse a node's left subtree, traverse the right subtree, and then visit the node.

- 1 traverse the left subtree
- 2 traverse the right subtree
- 3 visit the node

Given an expression tree for  $2 + 3 * 4$ , what is the result of a postorder traversal?

How would you change this algorithm for a  $k$ -ary tree?

How would you implement a traversal in reverse order?

# Level-Order Traversal

In a *level-order* or *breadth-first traversal* we visit the nodes in order of increasing depth and, among the nodes of the same depth, in left-to-right order.

Notice that a level-order traversal makes sense for  $k$ -ary trees, not just for binary trees. For a level-order traversal, we need a queue.

# Level-Order Traversal

For simplicity, assume a binary tree:

```
# Notice that this is a class method, not an instance method.
def levelorderAux(node):
    """Do a level-order traversal from the node."""
    q = MyQueue()
    q.enqueue( node )
    while not q.isEmpty():
        next = q.dequeue()
        print( next.getValue(), end=" ")
        if next.getLeft():
            q.enqueue ( next.getLeft() )
        if next.getRight():
            q.enqueue ( next.getRight() )
    return

def levelorder(self):
    if self.isEmpty():
        print("The tree is empty.")
    else:
        BinaryTree.levelorderAux(self._root)
    print("\n")
```

How would you change this for a  $k$ -ary tree?

# Testing our Traversals

```
>>> from Trees import *
>>> expr = "2 3 * 4 + 2 * 5 6 + *"
>>> tree = postfixToTree( expr )
>>> tree.getRoot().getValue()
'*'
>>> tree.inorder()
2 * 3 + 4 * 2 * 5 + 6

>>> tree.inorderWithParens()
( ( ( ( 2 * 3 ) + 4 ) * 2 ) * ( 5 + 6 ) )

>>> tree.preorder()
* * + * 2 3 4 2 + 5 6

>>> tree.postorder()
2 3 * 4 + 2 * 5 6 + *

>>> tree.levelorder()
* * + + 2 5 6 * 4 2 3
```

# Performance: Preorder, Inorder, Postorder

Assuming **visit** runs in constant time, then on a binary tree with  $n$  nodes, all three of these traversals run in time:

$$T(1) = O(1)$$

$$T(n) = T(n_l) + T(n_r) + O(1), \text{ otherwise.}$$

where  $n_l$  and  $n_r$  are the numbers of nodes in the left and right subtrees respectively.

By induction, we can easily show that on a binary tree with  $n$  nodes, all three traversals run in time

$$T(n) = O(n).$$

Assume that `visit` runs in constant time, and all of the queue operations have been implemented to run in constant time (which we know is possible). Then, on an ordered tree with  $n$  nodes, the level-order traversal also runs in time:

$$T(n) = O(n).$$

This fact follows from the observation that the while loop is executed once for each node.

# Evaluate an Expression Tree

Suppose you have a legal expression tree. How would you evaluate it?

Evaluation is clearly a recursive process. You know the value at any node if you can *recursively* compute the values of its left and right subtrees.

- 1 What are the base cases?
- 2 What are the recursive cases?
- 3 How do you know that the recursion terminates?

# Evaluate a Tree

```
def evalTreeAux( node ):  
    if node.isLeafNode():  
        return int( node.getValue() )  
    else:  
        return applyOp( node.getValue(),  
                        evalTreeAux( node.getLeft() ),  
                        evalTreeAux( node.getRight() ) )  
  
def evalTree( tree ):  
    if tree.isEmpty():  
        print ( "Tree is empty" )  
        return None  
    else:  
        return evalTreeAux( tree.getRoot() )
```

How would you add variables to our expression tree?

# Running the Evaluator

```
>>> expr = "2 3 * 4 + 2 *"  
>>> tree = postfixToTree( expr )  
>>> tree.inorder()  
2 * 3 + 4 * 2  
  
>>> evalTree( tree )  
20  
>>> tree.inorderWithParens()  
( ( ( 2 * 3 ) + 4 ) * 2 )  
  
>>> tree.postorder()  
2 3 * 4 + 2 *  
  
>>> tree.preorder()  
* + * 2 3 4 2
```

# Binary Search Tree

A *binary search tree* is a binary tree in which each node holds an entry, and the keys in each entry satisfy the following *binary search tree property*:

*Let  $y$  be a node with key  $k_y$  in a binary search tree. If  $x$  is a node with key  $k_x$  in the left subtree of  $y$ , then  $k_x \leq k_y$ . If  $z$  is a node with key  $k_z$  in the right subtree of  $y$ , then  $k_y \leq k_z$ .*

How would you print out the keys in sorted order?

The interface of a binary search tree is the same as a binary tree, with some additional functionality:

- searching
- insertion
- remove an element
- findMax, findMin (maybe)

For some purposes, it is also reasonable to store in each node a pointer to its parent.

# Searching in a Binary Tree

To find a particular key value in a binary search tree, start at the root and trace a path downward in the tree guided by the binary search tree property.

The search is easily coded with recursion, but an iterative version is usually more efficient in practice.

What do you think is the average case complexity of search?

# Searching (cont)

```
def inTreeAux(node, value):
    if node.getValue() == value:
        return True
    elif node.getValue() > value:
        if not node.getLeft():
            return False
        else:
            return BinarySearchTree.inTreeAux( node.getLeft(), value )
    else:
        if not node.getRight():
            return False
        else:
            return BinarySearchTree.inTreeAux( node.getRight(), value )

def inTree(self, item):
    if self.isEmpty():
        return False
    else:
        return BinarySearchTree.inTreeAux( self.getRoot(), item )
```

New nodes are always inserted as leaves. We perform a search to find the node that should be the parent of the new node. The new node is then added as a leaf.

What do you think is the average case complexity of search?

How would you handle the desire to have multiple occurrences of an item in a tree? Actually there are several methods:

- Just have multiple nodes with that value in the tree.
- At each node maintain a count of the times that value occurs.

# Insertion (cont)

```
def insertAux( node, item ):  
    if node.getValue() > item:  
        if not node.getLeft():  
            newNode = BinaryTreeNode( item )  
            node.setLeft( newNode )  
        else:  
            BinarySearchTree.insertAux( node.getLeft(), item )  
    else:  
        if not node.getRight():  
            newNode = BinaryTreeNode( item )  
            node.setRight( newNode )  
        else:  
            BinarySearchTree.insertAux( node.getRight(), item )  
  
def insert( tree, item ):  
    if tree.isEmpty():  
        tree.setRoot( BinaryTreeNode( item ) )  
    else:  
        BinarySearchTree.insertAux( tree.getRoot(), item )
```

# Running Our Code

```
>>> from Trees import *
>>> bst = BinarySearchTree()
>>> bst.insert(6)
>>> bst.insert(8)
>>> bst.insert(4)
>>> bst.inorder()
4 6 8

>>> bst.insert(12)
>>> bst.inorder()
4 6 8 12

>>> bst.inTree(4)
True
>>> bst.inTree(5)
False
>>> bst.inTree(12)
True
>>> bst.insert(1)
>>> bst.inorder()
1 4 6 8 12
```

To remove a node, we assume that we have a pointer to the node to remove. Then there are three cases:

- 1 If the node is a leaf, simply remove it.
- 2 If the node has only one child, simply replace the node with its child.
- 3 If the node has two children, then replace the node with its successor (node with next largest key) and remove the successor.

How do we find the successor of a node? What structural property is guaranteed to hold for the successor?

Notice that you can also replace a node with its predecessor. What would that look like?

# Try It Yourself

Try to write the code to:

- Remove an item from a binary search tree;
- Find the maximum / minimum element of a binary search tree.

# Lazy Deletion

An alternative strategy is to leave the value in the tree and mark it as deleted.

This is a pretty good strategy for binary trees, even if there are quite a few deletions. *Why?*

Suppose half of the items in the tree have been deleted. *What penalty does this impose on the performance of the various operations in comparison to the strategy of actually removing them?*

For all three operations—**inTree**, **insert**, and **remove**—we follow a single path from the root down to a leaf. Thus, all three operations run in time  $O(h)$  where  $h$  is the height of the tree.

How does the height of the tree relate to the number of nodes in the tree?

Why might the worst case behavior be much worse than the average case behavior for tree-based algorithms?

- In the worst case, we have  $h = n - 1 \in O(n)$ . This is the case if entries are inserted in sorted order by key.
- In the best case, we have  $h = \lfloor \log(n) \rfloor \in O(\log(n))$ . This case occurs if the tree is “balanced.”
- If the keys are inserted in random order, then the expected height is  $O(\log(n))$ .
- To maintain the expected logarithmic height in the presence of removals, when the node to be removed has two children, randomly pick between the successor and predecessor.

Can we do anything to assure that we always have the best case?

# Balanced Trees

Suppose you insert the following elements in order into a binary tree: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10.

The result is effectively a linked list and has the same performance. Even if a tree is initially “balanced,” additional insertions and deletions may leave it unbalanced and destroy the logarithmic behavior that makes binary trees desirable.

We can alleviate this problem by rebalancing the tree:

- 1 occasionally, whenever it becomes seriously unbalanced;
- 2 systematically, whenever we insert a new element into the tree;
- 3 systematically, whenever we access an element in the tree.

The most popular schemes follow approaches 2 or 3. (AVL trees, red-black trees, B-trees)

# TreeSort

You can derive a pretty efficient sorting algorithm (in the average case) if you use a binary search tree.

```
def TreeSort(lst):
    if lst == []:
        return []
    tree = BinarySearchTree()
    for i in lst:
        tree.insert(i)
    tree.inorder()
```

```
>>> lst = [ 9, 4, 6, 3, 1, 7, 4, 12, 0, -2 ]
>>> TreeSort (lst)
-2 0 1 3 4 4 6 7 9 12
```

What is the average complexity of this algorithm? This algorithm is  $O(n \log(n))$ . Each insertion takes  $\log(n)$  effort, and there are  $n$  of them.