Week 8 - Binary Search Trees

Updated October 29, 1:30pm

This week, we're going to learn about a new data structure called the **Binary Search Tree**. This is a basic type that's useful for storing *sorted* data, and is the basis for more advanced, efficient data structures you'll learn about in CSC263.

Last week we talked about trees, a very general data structure; then, on your exercise, you saw *binary trees*, a restricted type of tree where every node has at most two children. You might have wondered why we introduced this; indeed, just restricting the number of subtrees at each level to two doesn't seem to simplify any of the tree methods.

However, with one additional restriction on the *content* of the nodes rather than overall tree structure, binary trees become immensely useful!

Context: Searching for a Needle in a Haystack

Consider the general problem of searching for a particular item in a data structure. For lists, the obvious iterative algorithm (for both arrays and linked lists) is to loop through all items in the list and stopping when the item is found. The recursive variation you wrote for `LinkedListRec` follows the same basic idea. In the worst case, when the item is not in the list, all items must be checked, making this a *linear time approach* (if the length of the list doubles, so does the time taken).

In your lab last week, you implemented `__contains__` for trees, and it turns out that because in the worst case each subtree must be checked, this algorithm still takes linear time. So just switching from lists to trees isn't enough to do better!

However, we've also seen that if an *array* is sorted, we can use **binary search** to greatly improve the efficiency of searching. But because this is still based on arrays, we suffer the same drawbacks for insertion and deletion. So the question is: can we achieve **fast search, insertion, and deletion** all at once? Yes we can!

Binary Search Trees: Definitions

A node in a binary tree has the **binary search tree property** if its value is greater than or equal to all node values in its left subtree, and less than all node values in its right subtree.

A binary tree is a **binary search tree** if *every* node in the tree satisfies the binary search tree property (note that it's very possible in general for some nodes to satisfy this but not others).

Binary search trees naturally represent *sorted data*, making them extremely useful in doing operations like searching for an item; but unlike sorted Python lists, they can be much more efficient at insertion and deletion while keeping the data sorted!

BST Basics

Remember that the structure of a binary search tree is identical to that of a binary tree. Our code for the `BinarySearchTree` class will start off heavily based on `BinaryTree`.

```
class BinarySearchTree:
    def __init__(self, root=EmptyValue):
        self.root = root    # root value
        if self.is_empty():
            # Set left and right to nothing,
            # because this is an empty binary tree.
            self.left = None
            self.right = None
        else:
            # Set left and right to be new empty trees.
            # Note that this is different than setting them to None!
            self.left = BinarySearchTree()
            self.right = BinarySearchTree()

    def is_empty(self):
        return self.root is EmptyValue
```

We'll talk about *mutating* BST methods next lecture; for now, just assume that we can do things like insert new values into the tree. Instead, we'll focus today on non-mutating methods. The most important of these is probably `__contains__`.

Searching a BST

Recall that the key insight of the binary search algorithm is that by comparing the target item with the *middle* of the list, we can immediately cut in half the remaining items to be searched. An analogous idea holds for BSTs.

For general trees, the standard search algorithm is to compare the `item` against the root, and then search in *each* of the subtrees until either the item is found, or all the subtrees have been searched. In the worst case, when `item` is not in the tree, *every node* must be searched.

In stark contrast, for BSTs *the initial comparison to the root tells you which subtree you need to check*. That is, only one recursive call needs to be made, rather than two!

```
def __contains__(self, item):
    if self.is_empty():
        return False
    elif item == self.root:
        return True
    elif item < self.root:
        return self.left.__contains__(item)
    else:
        return self.right.__contains__(item)
```

Efficiency

(Note: we didn't get to this in lecture, but will return to this in detail later in the course!)

We claimed earlier that binary search trees allow for efficient searching, similar to the binary search in a sorted list. Why is this the case?

Suppose that the binary tree is very balanced: for every subtree, roughly half its nodes are in its left subtree, and half its nodes are in its right subtree. If this is true, then at every level of recursion, roughly half the elements are removed, just like with binary search. In this case, if there are n nodes in the tree, there are only $log_2 n$ recursive calls made in total.

However, now suppose the tree is very unbalanced: all of its nodes are on one side (*just like a linked list...*). Now, starting at the root and working our way down doesn't lead to any savings at all - in the worst case, we still end up checking every node!

It is a common theme among tree operations that the amount of time an algorithm takes depends on the *height* of the tree rather than simply the number of nodes. This makes sense intuitively, because recursion takes us *down* the tree, and the base case is reached when we've reached the bottom of the tree, making one recursive call per level.

Thus even binary search trees by themselves are not necessarily ideal: they're vulnerable to being very imbalanced, and hence not actually offering efficient search. In CSC263, you'll explore different ways of improving on the basic BST implementation to *guarantee* some sort of "balanced" structure, thereby gaining efficiency.

David Liu (liudavid at cs dot toronto dot edu)
Come find me in BA4260
Site design by Deyu Wang (dyw999 at gmail dot com)