BST Mutation

Updated October 29, 2pm

The really neat thing about Binary Search Trees is not that they can support efficient search - sorted lists achieve the same thing - it's that you can add and remove nodes in the tree efficiently as well. We'll see how to do this in today's lecture.

Insertion

Let's start with insertion. As with general tree insertion, the basic idea is the same: to avoid changing the contents of the tree too much, we will recurse down the tree until we reach an empty spot to insert the new leaf into.

However, while with general trees we did this randomly, here we must preserve the *binary search tree property*, and should determine where the new item should go. It is a somewhat intuitive fact that for a starting BST and new item, there is only *one leaf position* where the the item can go.

```
def insert(self, item):
    if self.is_empty():
        # Make new leaf node
        self.root = item
        self.left = BinarySearchTree()
        self.right = BinarySearchTree()
    elif item <= self.root:
        self.left.insert(item)
    else:
        self.right.insert(item)
```

Is this efficient? As with search, the worst-case number of recursive calls made depends on the height of the tree: if the tree is balanced, great! If not, this is just as inefficient as adding an element into a list.

Some students pointed out during lecture that there are other possibilities for inserting a new item other than making it into a leaf, and indeed, this is true! However, alternate approaches are more complex because they involve changing more of the tree structure, and so are beyond the scope of this course.

Deletion

With deletion, matters get a little more complicated. Once again, we'll start with the case of deleting the root node.

```
def delete_root(self):
    """ (BinarySearchTree) -> NoneType
    Delete root node of this tree.
    Raise EmptyBSTError if this tree is empty.
    """
```

One thing we might try is to set `self.root = EmptyValue`. Certainly this would remove the old value of the root, but this alone wouldn't change the structure of the tree! If we kept on doing this, eventually we'd have a tree full of empty nodes that we would have to recurse through, and this is extremely inefficient.

Instead, we're going to *replace* the root node with another value; and if you think about it long enough, there are **only two values** we could replace it with: the largest value in the left subtree, or the smallest value in the right subtree. Let's do one here: you'll do the other one in your lab.

Note the similarity to `extract_leaf` from a previous week: this function both *deletes* a value from the tree, and *returns* that value.

```
def extract_max(self):
    """ (BinarySearchTree) -> object
    Remove and return the maximum value in this tree.
    """

    if self.is_empty():
        raise EmptyTreeError
    elif self.right.is_empty():
        temp = self.root
        # Copy left subtree to self, because root node is removed.
        self.root = self.left.root
        self.right = self.left.right
        new_left = self.left.left
        self.left = new_left
        return temp
    else:
        return self.right.extract_max()
```

Now, our algorithm for deleting the root is pretty straightforward:

0. If the tree is empty, raise an error
1. Extract the maximum from the left subtree
2. Set the root value equal to that extracted value

```
def delete_root(self):
    if self.is_empty():
        raise EmptyBSTError
    else:
        self.root = self.left.extract_max()
```

And from this, we get a remarkably straightforward algorithm for deleting an item from a BST. Once again, note that we're guaranteed to make only *one* recursive call!

```
def delete(self, item):
    if not self.is_empty():
        if self.root == item:
            self.delete_root()
        elif item < self.root:
            self.left.delete(item)
        else:
            self.right.delete(item)
```

Wait! There's a problem. Test this code very carefully on all possible cases with tree

structure, until you find the error! Then, you'll fix this problem in your lab.

Side note: duplicate elements

A very subtle point was raised during lecture when we talked about `delete_root`. If we try to replace the root using the smallest element in the right subtree, *and* there is more than one copy of the smallest element in the right subtree, then we'll end up violating the "duplicates on the left side" invariant! (Draw a picture to visualize this.)

This does not rule out using the smallest element in the right subtree for all situations, and I'll leave it to you to think about:

- in which cases do you absolutely *need* to use this strategy
- how to detect if the strategy will fail
- how to fix the violation of the invariant
- alternate approaches to avoid using this strategy altogether

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