['Binary Search' + x for x in ['''', ' Trees']]
Francois's Announcements

Apologies for missing the lecture tonight: I've had to get back home to deal with a small emergency... But I leave you in good hands with Velian, who is a TA for the course and has also taught it before.

Project 1 test results will be uploaded to MarkUs sometime this week. If you have questions or concerns about the marking of your project, please be patient and have a look at the test results when they are available. Then, let me know if you still have a question or concern.

Note that markers were told to give 3/4 on categories where your solution was good, and 4/4 only for "exceptional" solutions.
Request for Note Takers: please see course forum for details. Accessibility Services requests volunteer note-takers for the course. This involves just taking notes as you normally would, except that you make them available to accessibility services. Students with a disability get the benefit of another student's notes, and you get the benefit of doing something good and receiving a certificate at the end of the course.
Test 2 takes place next week! It's on Wednesday (March 13th), **FROM 6:10PM TO 7:00PM** (note the change from the original time), in rooms EX 310 (last names F-S) and EX 320 (last names T-Z, A-E).

Topics: linked lists and trees (and everything related, including our old friend, recursion...)

The test will be followed by 1 hour of lecture (7:10pm-8:00pm), back in our usual room.
Exercise 4: around 20 students are being investigated for plagiarism, because they submitted code that was CLEARLY not entirely their own.

Since I don't really have time to deal with this right now, I may give a "way out" for those students by providing everyone in the class with the opportunity to voluntarily withdraw their submission and accept a grade of 0 -- the alternative being to go through the process of being accused of plagiarism, and most likely suffering the penalties (a grade of 0 on E4, and additional penalties).

Watch for an announcement on the forum (and by email) in the next few days.
Binary Search
Binary Search

Question: is 31 in this sorted list?

Recurse on half the list

Recurse on half the list
Binary Search Trees
A **Binary Search Tree (BST)** is a special type of binary tree that adheres to the following property:

**BST Property:** All nodes in the left subtree of node with key $k$ have keys that are less than $k$. All nodes in the right subtree of node with key $k$ have keys that are greater than $k.$

Question: Is it enough to require that the root's key is greater than the left child's key and less than the right child's key?

Question: Is this the only BST for these keys?
Binary Search Trees (BST) can also be defined recursively.

A BST is composed of a root node with a BST containing only keys less than the root key on the left branch and a BST containing only keys greater than the root key on the right branch.
BST Operations

We will look at three basic BST operations:

**SEARCH** - determining whether a key is in the tree

**INSERT** - adding a new node to the tree without breaking the BST property

**DELETE** - removing an existing node from the tree without breaking the BST property
**BST Search**

We start at the root. (Where else?)

Our task for each node: compare the target value to the node's key.

If the target value is equal to the node's key, we have found our item and we return `True`.

If the target value is less than the node's key, we search for the value in the left subtree.

If the target value is greater than the node's key, we search for the value in the right subtree.

Finally, if the node doesn't exist, then we can't find the value and we return `False`.
BST Search

We start at the root. (Where else?)

Our task for each node: compare the target value to the node's key. and use it to decide which branch of the root to recurse on.

Target: 8

How will we know an item is NOT in the tree?
How would we actually implement this?
BST Insert

Inserting into a BST has a very important secondary purpose:

Do **NOT** mess up the BST property of the resulting tree.

We start at the root, and for every node:

- If `value < node.item`: insert to the left
- If `value > node.item`: insert to the right
- If `value == node.item`: do nothing! (this means our BST will only contain unique values!)

What if `node` is `None`?
If `node` is `None`, it means we have found the correct place for the value, but there is no node there yet, so we create the new node and return it up the chain. Let's code this up!
BST Deletion
Deletion

Deletion, like insertion, is closely tied to search.

We start at the root, and we go down the appropriate branches of the tree until we find the node we wish to delete.

Then, we have to alter the structure of the BST so that the node is no longer in the tree, making sure:

- the BST property is still satisfied
- we don't cut loose any more nodes than we have to
- we update all nodes' links correctly
Case 1: Leaf Node

Target: 9

Simple: delete the node
Case 1: Leaf Node

Target: 9

Does anyone foresee a problem deleting the node?
Deletion

The problem is that "deleting the node" really involves changing the node's parent's link to point to None. Once we've found the node to delete, we have no way of reaching its parent to change the appropriate child link.

**Possible solutions:**
- keep track of a parent for every node (increases data requirements by 33%, means more updating)
- perform a look-ahead search - instead of asking "Is this node's key equal to value?", ask "is this node's child's key equal to value?"
- Use the return value of the recursive call to reflect changes in the subtree
Case 1: Leaf Node

Make the recursive `delete` function return the root of the subtree it's working on. Then, when calling `delete`, take the result of the call and update your links appropriately.

Target: 9
Case 2: One Subtree

Target: 3

Replace node with the root of its subtree.
Case 2: One Subtree

Target: 3

Replace node with the root of its subtree.
Case 3: Two Subtrees

Target: 14

We should swap the node's value with another node in its subtree that would be easier to delete.

The node we choose should be such that we have to do the least amount of work to restore BST properties.

E.g. If we swap with 1, we have five "problem nodes" which are now to the left of a smaller node.
Case 3: Two Subtrees

Target: 14

We should swap the node's value with another node in its subtree.

What's the best node to do this with?

In the left subtree,
we should pick the largest node.

In the right subtree,
we should pick the smallest node.

These are called the in-order predecessor and the in-order successor.

Question: at most how many subtrees do either of these nodes have? Why?
Case 3: Two Subtrees

Target: 14

Case 3.1. In-order Predecessor

Find L, the largest node in the left subtree.

Copy L's value into the node to delete.

Delete L (reduced to case 1 or 2)
Case 3: Two Subtrees

Target: 14

Case 3.1. In-order Predecessor

Find L, the largest node in the left subtree.

Copy L's value into the node to delete.

Delete L (reduced to case 1 or 2)
Case 3: Two Subtrees

Target: 14

Case 3.1. In-order Predecessor

Find L, the largest node in the left subtree.

Copy L's value into the node to delete.

Delete L (reduced to case 1 or 2)
Case 3: Two Subtrees

Target: 14

Case 3.2. In-order Successor

Find R, the smallest node in the right subtree.

Copy R's value into the node to delete.

Delete R (reduced to case 1 or 2)