Binary Search Trees

CSC148, INTRODUCTION TO COMPUTER SCIENCE
DAVID LIU
A Binary Search Tree is a “sorted” tree

Every item is \( \geq \) all items in its left subtree, and \( \leq \) all items in its right subtree.
def __contains__(self, item) -> bool:
    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)
But not always!

def items(self) -> List:
    if self.is_empty():
        return []
    else:
        return (self._left.items() +
                [self.root] +
                self._right.items())
Representation invariants are key!

If `self._root` is not `None`, then `self._left` and `self._right` are `BinarySearchTree`s.

If you know that the BST is not empty, you **never** need to check if `self._left` or `self._right` are `None`. 
Deleting from a BST

HINT: PULL OUT THE TREE DELETION WORKSHEET FROM LAST WEEK
```python
def delete(self, item: Any) -> None:
    """Remove *one* occurrence of <item> from this BST.

    Do nothing if <item> is not in the BST.
    """
```
First, need to find the item
Introducing Assignment 2!

YAY!
Autocompletion
Autocompletion for melodies
Back to efficiency

WHY SHOULD WE CARE ABOUT BINARY SEARCH TREES?
The Multiset ADT (search, insert, delete)

For a **sorted list** with $n$ items...

- search is fast: $O(\log n)$ worst case, because of binary search
- insert and delete can be slow, if inserting/removing from the *front* of the list – $O(n)$ in the worst case
The Multiset ADT (search, insert, delete)

For a **general tree** with $n$ items...

```python
for subtree in self.subtrees:
    if subtree.__contains__(item):
        return True
return False
```
The Multiset ADT (search, insert, delete)

For a **general tree** with $n$ items...

- insert can be fast, if you insert as a child of the root – $O(1)$
- search and delete can be slow, since you might need to check every item in the tree – $O(n)$ in the worst case
Worst case running times so far...

<table>
<thead>
<tr>
<th>operation</th>
<th>Sorted List</th>
<th>Tree</th>
<th>Binary Search Tree</th>
</tr>
</thead>
<tbody>
<tr>
<td>search</td>
<td>$O(\log n)$</td>
<td>$O(n)$</td>
<td></td>
</tr>
<tr>
<td>insert</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
<td></td>
</tr>
<tr>
<td>delete</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td></td>
</tr>
</tbody>
</table>
Binary search trees

In a binary search tree, each Multiset operation’s worst-case running time is proportional to the height of the tree.

\[
\log n \leq h \leq n
\]

<table>
<thead>
<tr>
<th>operation</th>
<th>Sorted List</th>
<th>Tree</th>
<th>Binary Search Tree</th>
</tr>
</thead>
<tbody>
<tr>
<td>search</td>
<td>O(log (n))</td>
<td>O((n))</td>
<td>O(h)</td>
</tr>
<tr>
<td>insert</td>
<td>O((n))</td>
<td>O(1)</td>
<td>O(h)</td>
</tr>
<tr>
<td>delete</td>
<td>O((n))</td>
<td>O((n))</td>
<td>O(h)</td>
</tr>
</tbody>
</table>
BST height vs. size

A binary search tree of size $n$...

- has a maximum height of $n$: $h \leq n$
- has a minimum height of (approximately) $\log n$: $h \geq \log n$

We say that a BST is balanced if its left and right subtrees have roughly equal heights, and these subtrees are also balanced.

Balanced BSTs have height $\approx \log n$. 
## Efficiency, final word... for CSC148!

<table>
<thead>
<tr>
<th>operation</th>
<th>Sorted List</th>
<th>Tree</th>
<th>BST</th>
<th>Balanced BST</th>
</tr>
</thead>
<tbody>
<tr>
<td>search</td>
<td>O(log <em>n</em>)</td>
<td>O(<em>n</em>)</td>
<td>O(<em>h</em>): O(<em>n</em>)</td>
<td>O(<em>h</em>): O(log <em>n</em>)</td>
</tr>
<tr>
<td>insert</td>
<td>O(<em>n</em>)</td>
<td>O(1)</td>
<td>O(<em>h</em>): O(<em>n</em>)</td>
<td>O(<em>h</em>): O(log <em>n</em>)</td>
</tr>
<tr>
<td>delete</td>
<td>O(<em>n</em>)</td>
<td>O(<em>n</em>)</td>
<td>O(<em>h</em>): O(<em>n</em>)</td>
<td>O(<em>h</em>): O(log <em>n</em>)</td>
</tr>
</tbody>
</table>