6.7 Binary Search Trees and Running Time

- The implementation of `__contains__`, `insert`, and `delete` has the same structure (one recursive call inside the recursive step)
- Max number of recursive calls == height of the tree + 1 (maximum of a set of possible running times)
  - The extra call comes because our implementation also recurses into the empty subtree of a leaf

LOOKING AT _ _contains_ _:

```
def __contains__(self, item: Any) -> bool:
    """Return whether <item> is in this BST.
    """
    if self.is_empty():
        return False
    else:
        if item == self._root:
            return True
        elif item < self._root:
            return item in self._left   # or, self._left.__contains__(item)
        else:
            return item in self._right  # or, self._right.__contains__(item)
```

- All lines except the recursive call run in constant time
- Total running time is proportional to the number of recursive calls made.
- Max # Recursive calls roughly height of the tree:
  - **O(n) , n = the height of the tree**
- **Worst case:** when the item we are looking for makes us recurse down to the deepest level and search one of its empty subtrees
  - O(n) , n = the height of the tree
- **Best case:** when we search for the root number in the binary search Tree.
  - O(1) , independent of Tree's height

WORST-CASE VS BEST-CASE RUNNING TIME:

- Running time of function/method depends on:
  - Size of inputs
  - For fixed input size: searching for root item of BST vs searching for an item which is very deep in BST
- Worst-case, Best-case should make sense for any input size!

WORST-CASE RUNNING TIME:

- Function WC(n)
- Maps input size n to the *maximum* possible running time for all inputs of size n.

- Family of inputs (each input results in max running time for its size.)

BEST-CASE RUNNING TIME:

- Function BC(n)
- Maps input size n to the *minimum* possible running time for all inputs of size n.
- Description for a family of inputs

TREE HEIGHT AND SIZE:

- Searching through an unsorted list takes O(n), n is the size of the list
- This doesn't work for BST:
  - The height of the tree can be much smaller than its size
- Considering a Binary Search Tree with n items:
  - Height can be as large as n
  - Height can be as small as log(n)
- Three Collection operations (search, insert, delete):
  - Worst case running time : $O(h) = O(\log n)$, h is height and n is size
- Since we can't always guarantee the algorithm will be logarithmic we can't really guarantee its efficiency yet :)

- Recall:
  - $\log_a x = y \iff a^y = x$
  - Example: $2^5 = 32 \iff \log_2 32 = 5$
  - $\log_2 n$, often known in CS as $\log n$
    - After all, base 2 is our favorite base in CS .. :)

- In a binary search tree, each Multiset operation's worst-case running time is proportional to the height h of the tree (where $\log n \leq h \leq n$).

operation	Sorted List	Tree	Binary Search Tree
search	$O(\log n)$	$O(n)$	**$O(h)$**
insert	$O(n)$	$O(1)$	**$O(h)$**
delete	$O(n)$	$O(n)$	**$O(h)$**

AVL trees ...

operation	Sorted List	Tree	BST	Balanced BST
search	$O(\log n)$	$O(n)$	$O(h)$: $O(n)$	$O(h)$: **$O(\log n)$**
insert	$O(n)$	$O(1)$	$O(h)$: $O(n)$	$O(h)$: **$O(\log n)$**
delete	$O(n)$	$O(n)$	$O(h)$: $O(n)$	$O(h)$: **$O(\log n)$**