Introduction - Insertion and Search

In this lesson, you will learn how to implement Binary Search Trees in Python and how to insert and search elements within them.

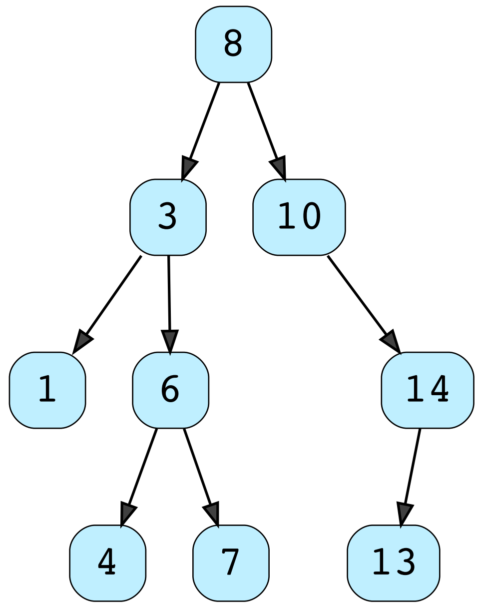
We'll cover the following

In this lesson, we will go over the binary search tree data structure. We will first cover the general idea of what a binary search tree is and how one may go about inserting data into this structure as well as how one searches for data. Once we cover the general idea, we will move over to the implementation of the binary search tree data structure in Python. We will construct two class methods that will implement the search and insertion algorithms.

A **binary search tree** is a tree data structure in which nodes are arranged according to the BST property which is as follows:

The value of the left child of any node in a binary search tree will be less than whatever value we have in that node, and the value of the right child of a node will be greater than the value in that node.

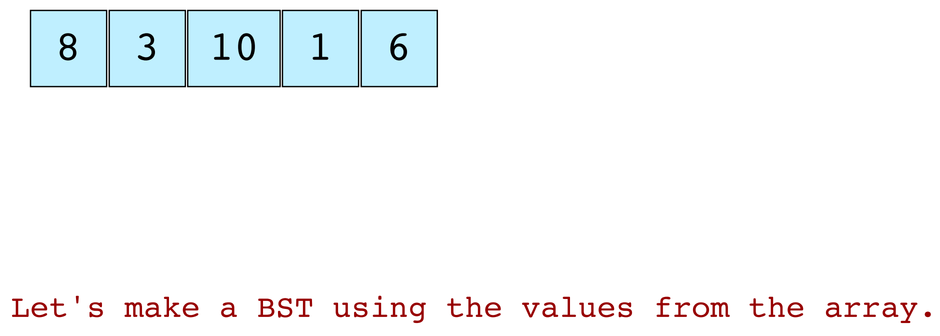
This type of pattern persists throughout the tree. Let's look at an example illustrated below:



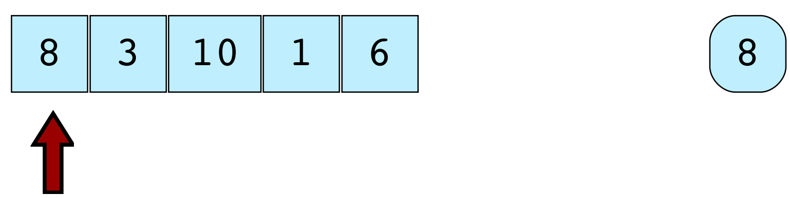
A Binary Search Tree

Insertion

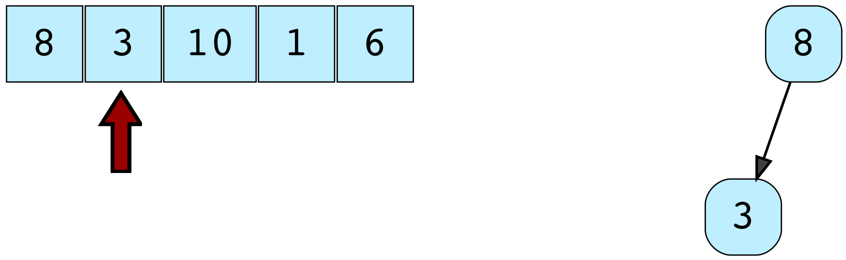
Let's see how we can insert elements in a binary search tree. In the illustration below, we have an array of elements where we want to insert all the elements one by one into the binary search tree. The way we go about insertion is to start from the root node and compare the new value to be inserted with the current node's value. If the new value is less than the value of the current node, then the new value must be inserted in the left subtree of the current node. On the other hand, if it's greater, it must be inserted in the right subtree to satisfy the BST property. We keep comparing the values as we traverse the tree and insert wherever we find a null position.


8
3
10
1
6
Let's make a BST using the values from the array.

1 of 6

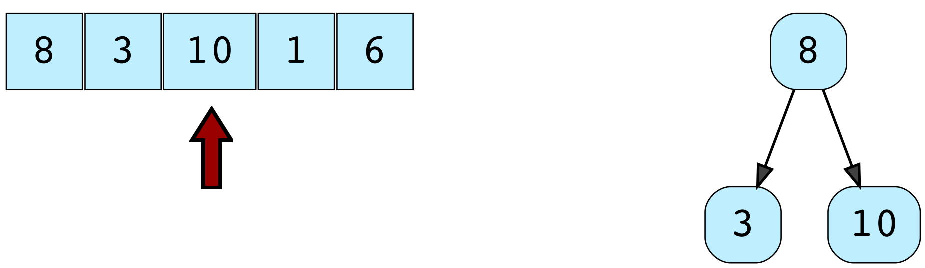

8
3
10
1
6
8

2 of 6



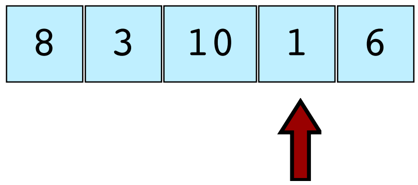
As 3 < 8, it is inserted
in the left subtree of 8.

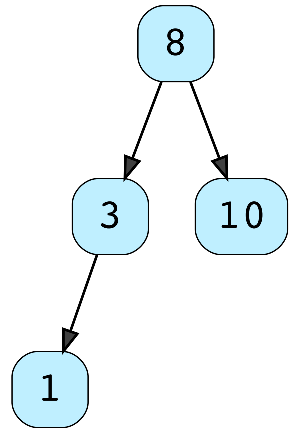
3 of 6



As 10 > 8, it is inserted
in the right subtree of 8.

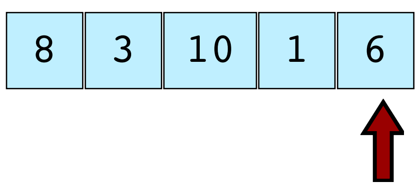
4 of 6

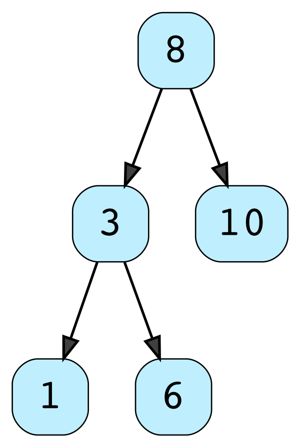




As 1 < 8, we traverse to the
left subtree of 8.
As 1 < 3, it is inserted
in the left subtree of 3.

5 of 6



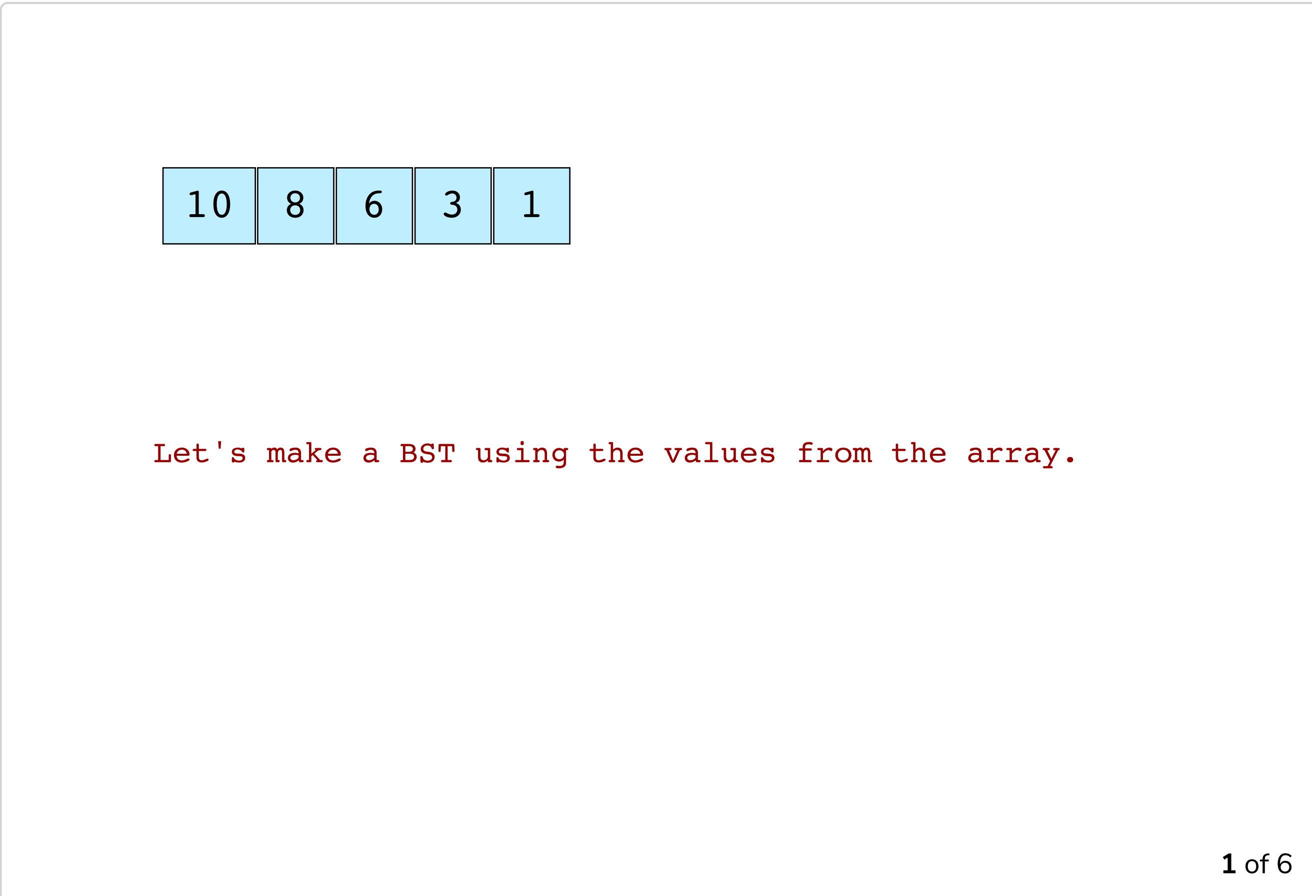


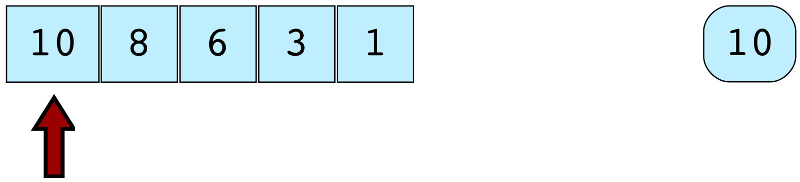
As 6 < 8, we traverse to the
left subtree of 8.
As 6 > 3, it is inserted
in the right subtree of 3.

6 of 6

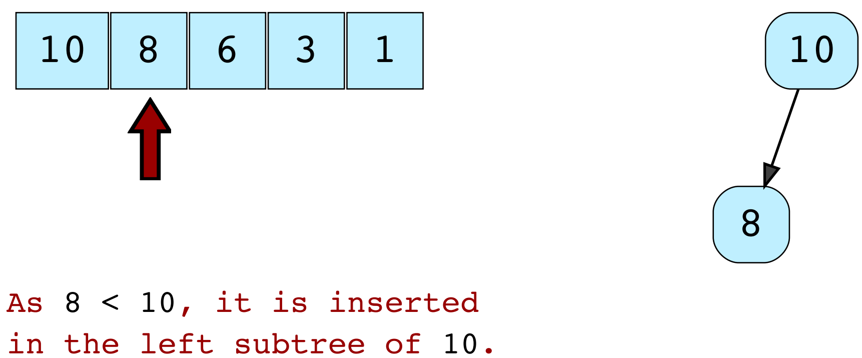
Insertion of a Reverse Sorted List

Let’s see what happens if we insert elements from a reverse sorted list one by one in a binary search tree.

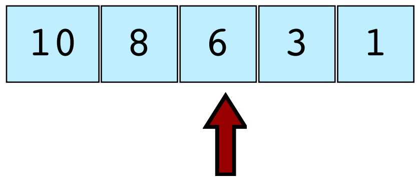


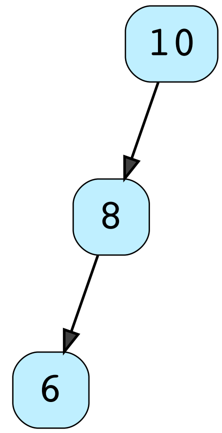

10
8
6
3
1
10

2 of 6


10
8
6
3
1
10
8
As 8 < 10, it is inserted
in the left subtree of 10.

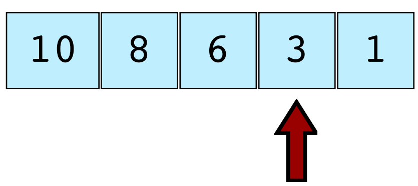
3 of 6

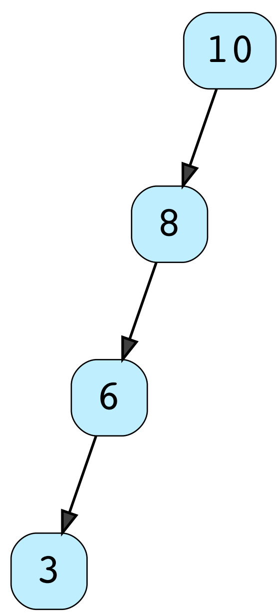




As 6 < 10, we traverse to the
left subtree of 10.
As 6 < 8, it is inserted
in the left subtree of 8.

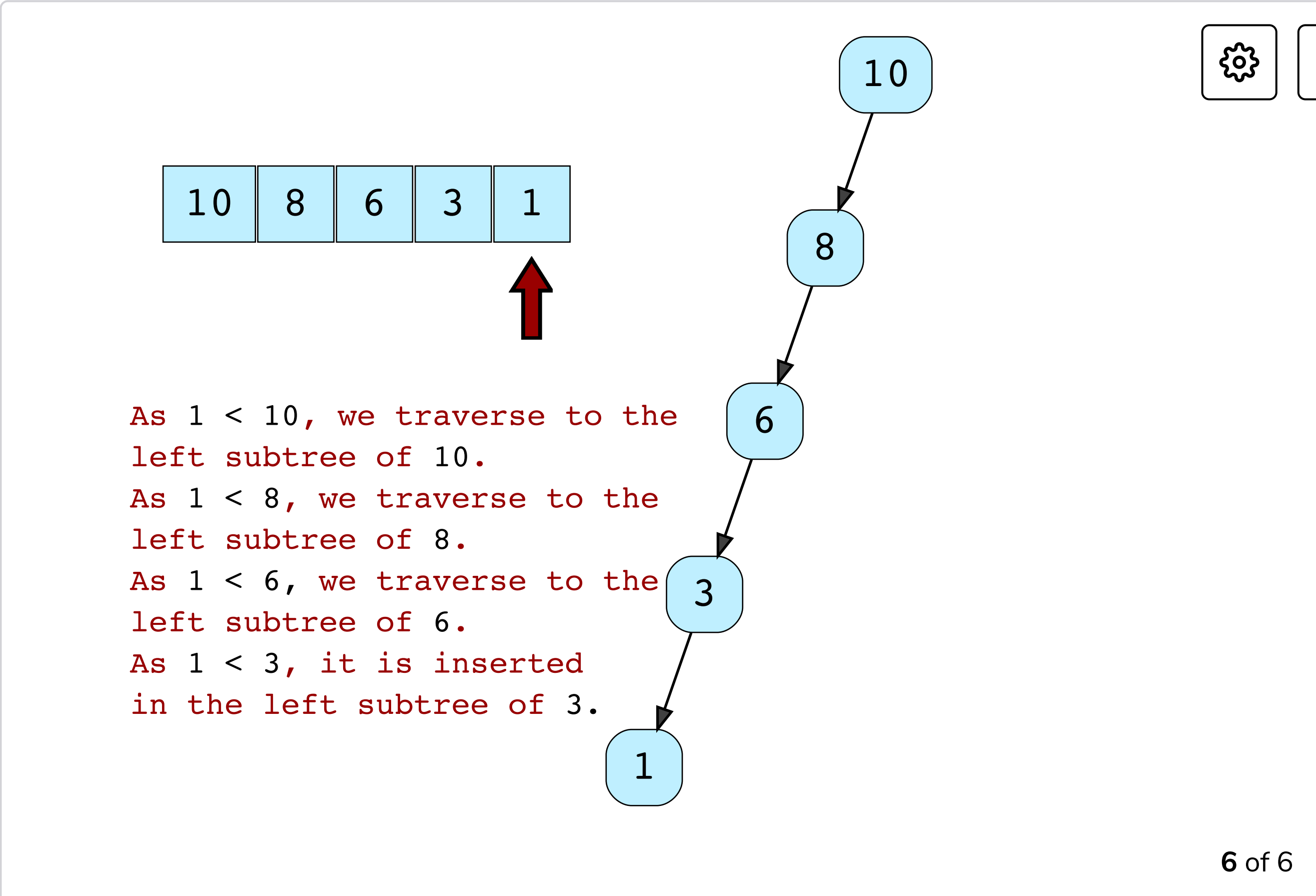
4 of 6





As 3 < 10, we traverse to the
left subtree of 10.
As 3 < 8, we traverse to the
left subtree of 8.
As 3 < 6, it is inserted
in the left subtree of 6.

5 of 6

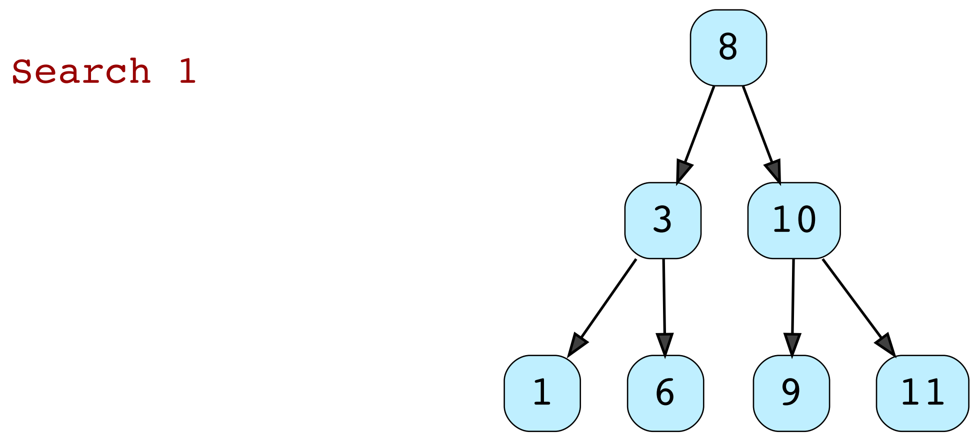


As you can see, we end up with a linear sort of a binary search tree. In such a case, if you want to find **node 1** in the above binary search tree, you might have to traverse down. However, if you have a structure other than the linear one, then you can substantially cut down the number of operations that you would have to do to find **node 1**. We will look at that sort of a structure when we cover searching. The linear binary search tree is a structure we want to avoid when we perform operations on binary search trees because it kills the purpose of having a binary search tree. If you're curious about how you can prevent getting the structure from reading data and forming a binary search tree out of it, then you can look into something called an AVL tree. An AVL tree rebalances the nodes so that you get a hierarchically spread-out structure instead of a linear one. However, that is beyond the scope of this lesson.

Search

We search using a similar approach as we used in the insertion in a binary search tree. Starting from the root node, we decide which subtree to traverse by comparing the value to be searched with the current node. Then we traverse to the appropriate subtree and discard the other subtree that does not contain the element we are searching for due to the BST property.

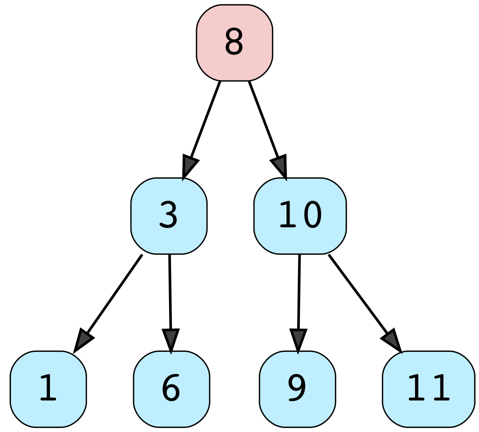
Have a look at the illustrations to get a fair idea:



1 of 4

Search 1

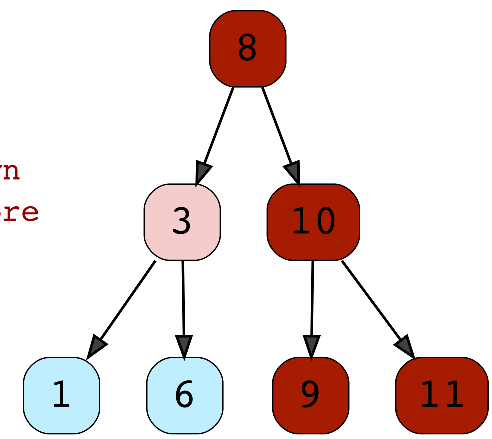
Start from the root node



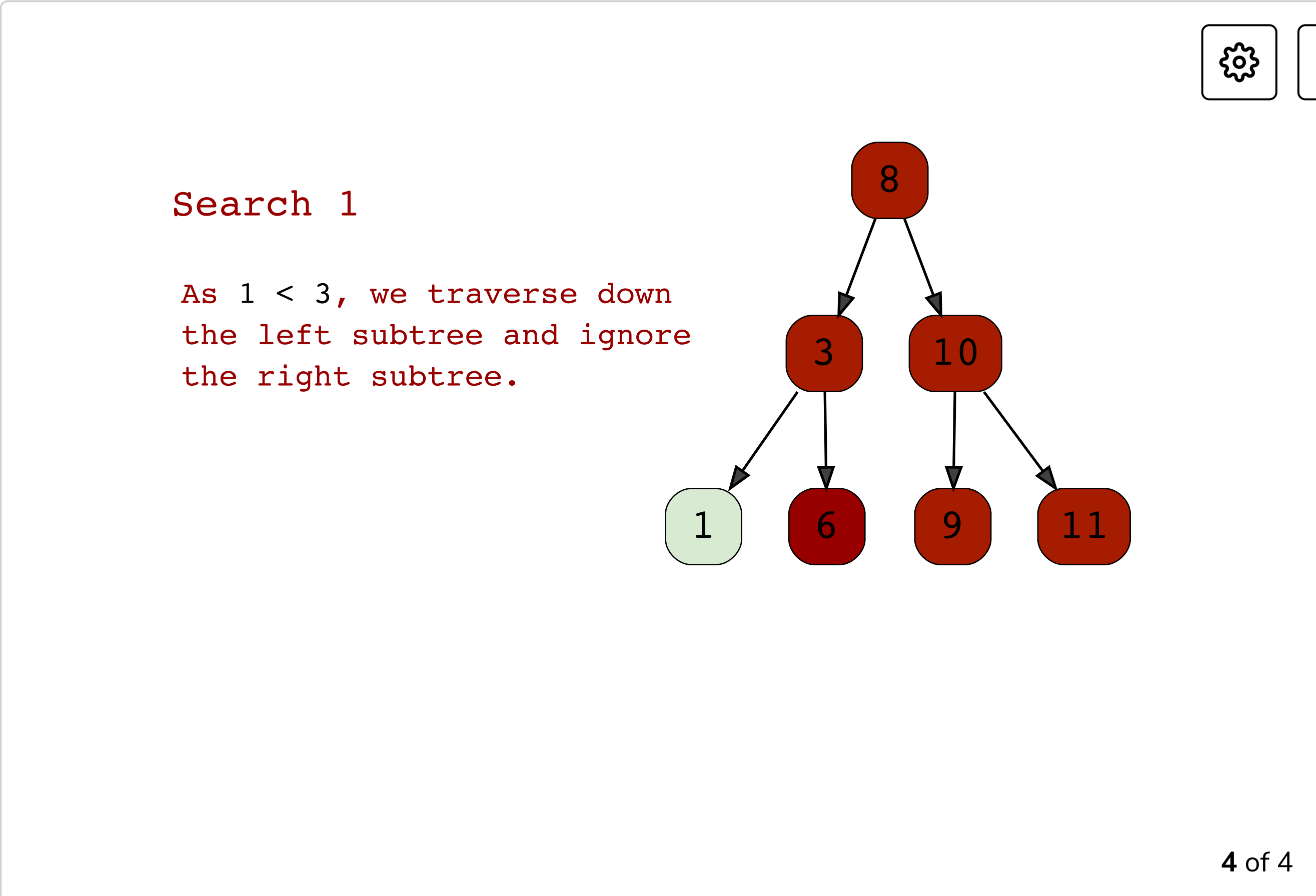
2 of 4

Search 1

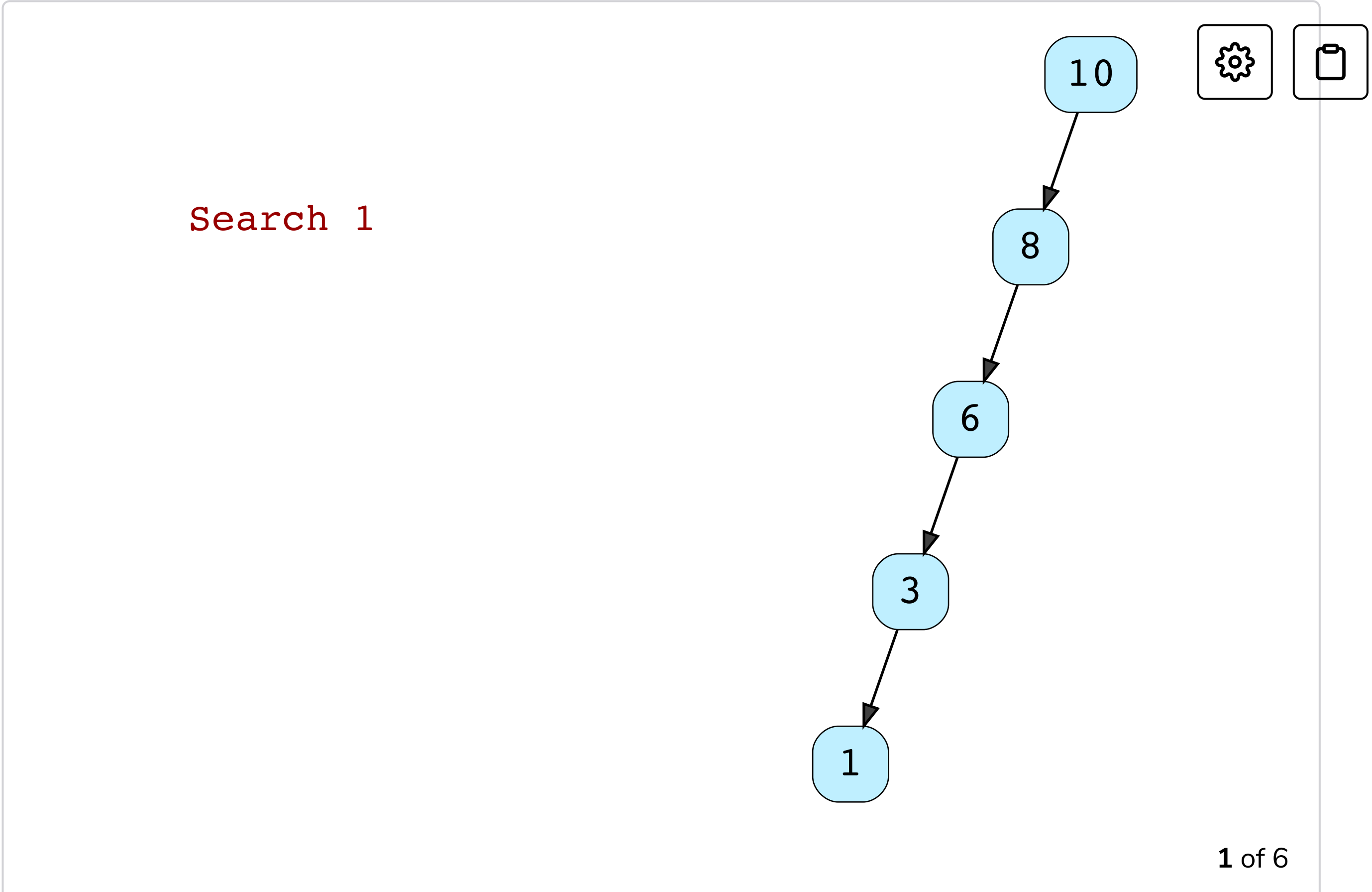
As 1 < 8, we traverse down the left subtree and ignore the right subtree.



3 of 4



Now let's also illustrate an example to search on a linear binary tree.



As you can see from the illustration above, we don't discard anything in a linear structure, so we have to traverse *all the nodes* to find the node we are looking for. This makes it an operation of $O(n)$ in the worst case. If we have a non-linear structure for a BST, the time complexity significantly improves to $O(logn)$.

This table summarizes the time complexities for a BST:

Algorithm	Average Case	Worst Case
Search	$O(logn)$	$O(n)$
Insert	$O(logn)$	$O(n)$

Algorithm	Average Case	Worst Case
Delete	$O(logn)$	$O(n)$

Now let's jump to the implementations in Python!

Implementation

The implementation of the `BST` class is similar to the implementation of the `BinaryTree` class. The `Node` class is exactly the same as the `Node` class in the Binary Trees chapter.

Have a look at the implementation below:

```
class Node(object):
  def __init__(self, data):
    self.data = data
    self.left = None
    self.right = None


class BST(object):
  def __init__(self, root):
    self.root = Node(root)
```

class Node and class BST

`insert`

Now we'll look into the implementation of `insert` operation in Python.

```
def insert(self, new_val):
  self.insert_helper(self.root, new_val)

def insert_helper(self, current, new_val):
```

```
5    if current.data < new_val:
6      if current.right:
7        self.insert_helper(current.right, new_val)
8      else:
9        current.right = Node(new_val)
10   else:
11       if current.left:
12         self.insert_helper(current.left, new_val)
13       else:
14         current.left = Node(new_val)
```

insert(self, new_val) and insert_helper(self, current, new_val)

The `insert` method on **line 1** invokes the helper function (`insert_helper`) on **line 2**, while also passing `self.root` and `new_val` into that method. In the `insert_helper` method, we have `current` and `new_val` which refer to the current node and the new value to be inserted. In order to find the appropriate location to insert `new_value` , we compare `current.data` to `new_value` . If `current.data` is less than `new_val` , then `new_val` must be placed somewhere in the right subtree. Therefore, we check on **line 6** if `current.right` is `None` . If it's not, this implies that we have to traverse the tree further and we recursively call `insert_helper` on **line** 7 and pass `current.right` and `new_value` to it. On the other hand, if `current.right` is `None` , then `current` must be a leaf node, and we have found an appropriate position to insert the `new_val` . Then we initialize `current.right` to a new `Node` by passing `new_val` to the constructor on **line 9**.

Let's talk about when `current.data` is greater than or equal to `new_val` . If that's the case, we repeat what we discussed above about the code in **lines 6-9**, but replace `current.right` with `current.left` as `new_val` must be placed in the left subtree of the current node.

`search`

Let's discuss the `search` method:

```
1  def search(self, find_val):
2    return self.search_helper(self.root, find_val)
3
4  def search_helper(self, current, find_val):
5    if current:
6      if current.data == find_val:
7          return True
8      elif current.data < find_val:
9          return self.search_helper(current.right, 
10     else:
11         return self.search_helper(current.left, f
```

search(self, find_val) and search_helper(self, current, find_val)

If you get the hang of the insert operation, then the `search` method is relatively simple to understand. Like the `insert` method, the `search` method invokes a helper function (`search_helper`) and passes the root node and the value to search (`find_val`) to the helper method. It also returns whatever is returned from the helper method.

In `search_helper`, we check on **line 5** if `current` is `None`. If it's not, then the execution jumps to **line 6** where the condition acts as a sort of base case to the recursive method. If the value of the current node (`current.data`) is equal to `find_val`, then we have found the node that we are supposed to search. `True` is returned on **line 7** to send a confirmation back to the caller method. However, if the condition on **line 6** does not evaluate to `True`, we check on **line 8** to see if `current.data` is less than `find_val`. If this evaluates to `True`, `find_val` must be in the right subtree of the current node to which we make a recursive call on **line 9**. Otherwise, if `current.data` is greater than `find_val`, we make a recursive call to the left subtree on **line 11**.

The entire code that we have implemented can be found below. Make a BST and insert elements to it to verify the `insert` and the `search` methods.

```
  def insert(self, new_val):
    self.insert_helper(self.root, new_val)
```

```python

  def insert_helper(self, current, new_val):
    if current.data < new_val:
      if current.right:
        self.insert_helper(current.right, new_val
      else:
        current.right = Node(new_val)
    else:
        if current.left:
          self.insert_helper(current.left, new_va
        else:
          current.left = Node(new_val)

  def search(self, find_val):
    return self.search_helper(self.root, find_val

  def search_helper(self, current, find_val):
    if current:
      if current.data == find_val:
          return True
      elif current.data < find_val:
          return self.search_helper(current.right,
      else:
          return self.search_helper(current.left,

bst = BST(10)
bst.insert(3)
bst.insert(1)
bst.insert(25)
bst.insert(9)
bst.insert(13)

print(bst.search(9))
print(bst.search(14))
```

Output 0.79s

```
True
None
```

Brace yourself for a challenge in the next lesson!

Back
Quiz

Next
Exercise: Checking the BST property

Mark as Completed

Report an Issue

Ask a Question
(https://discuss.educative.io/tag/introduction-insertion-and-search__binary-search-trees__data-structures-and-algorithms-in-python)

educative

Exercise: Checking the BST property

Challenge yourself with an exercise in which you'll have to check the BST property for a Binary Search Tree!

We'll cover the following

Problem

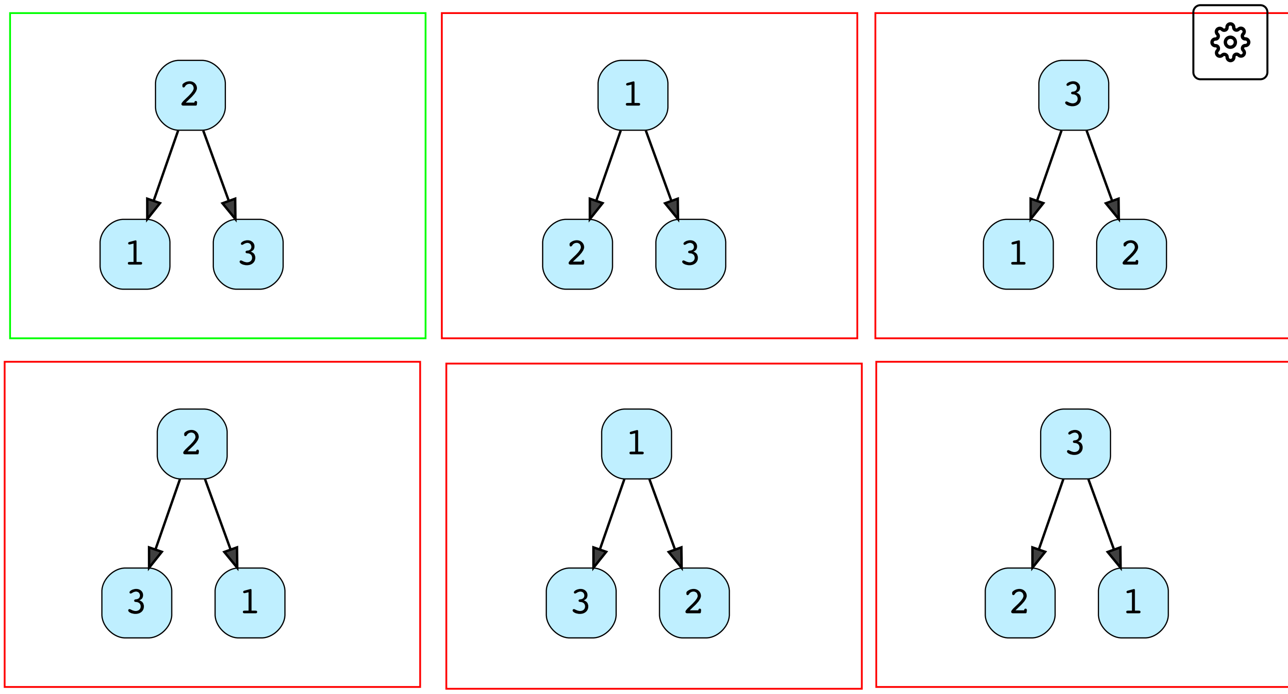
You are required to check and determine whether a tree satisfies the BST property. First of all, let's define the BST property.

BST Property

The BST property states that every node on the right subtree has to be larger than the current node, and every node on the left subtree has to be smaller than the current node.

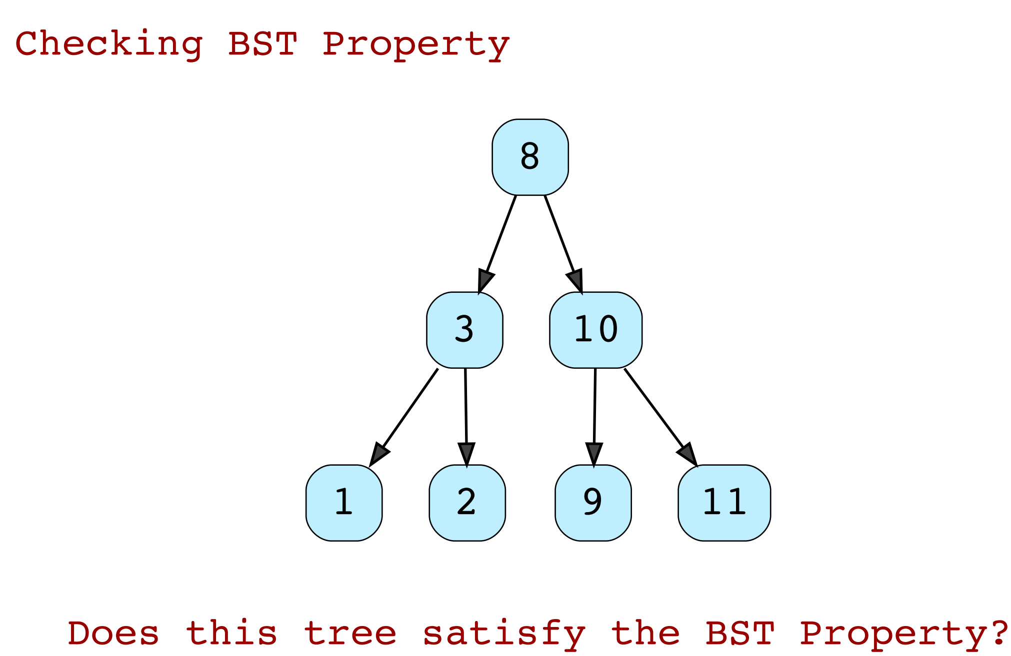
The binary search tree property (BST property) is a global property that every binary search tree must satisfy.

Below are some examples that show which trees satisfy the BST property:



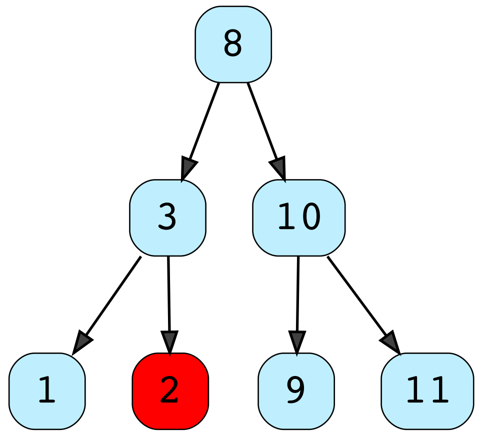
Examples

Here are some other examples where we check for the BST property:



1 of 6

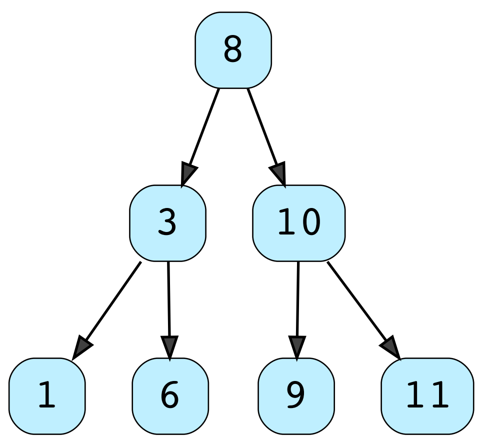
Checking BST Property



Does this tree satisfy the BST Property?
No! 2 is less than 3, so it can't be the right child of 3.

2 of 6

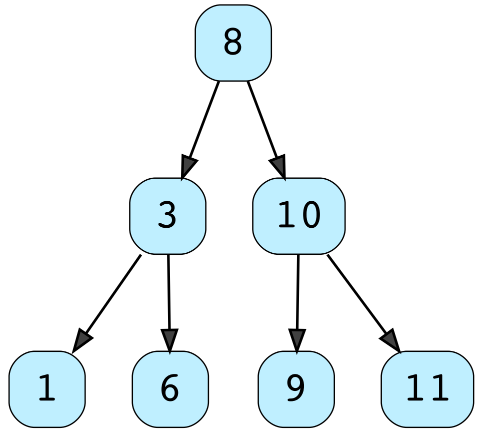
Checking BST Property



Does this tree satisfy the BST Property?

3 of 6

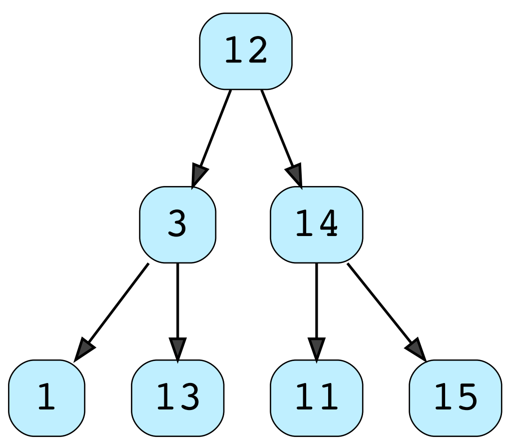
Checking BST Property



Does this tree satisfy the BST Property?
Yes!

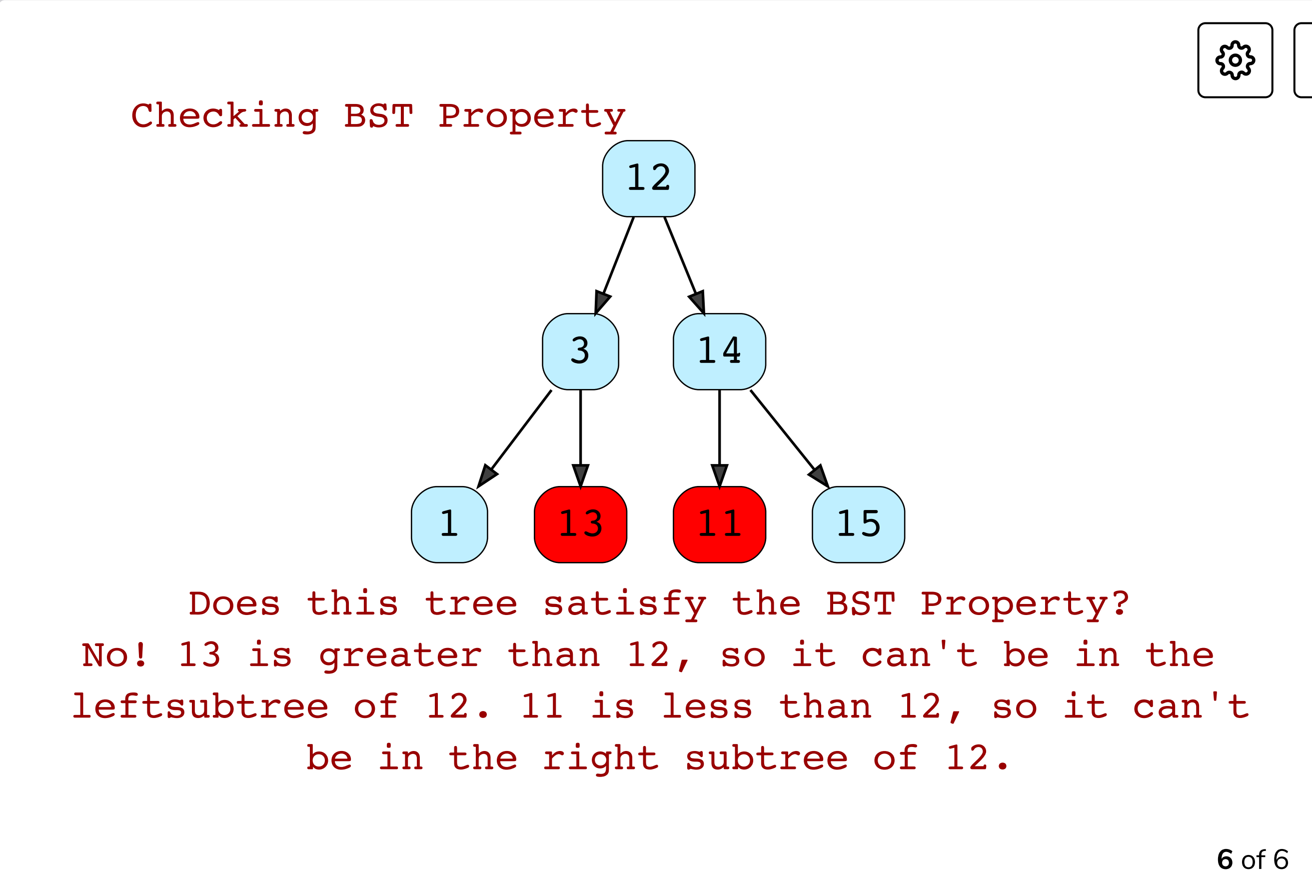
4 of 6

Checking BST Property



Does this tree satisfy the BST Property?

5 of 6



Coding Time!

In the code below, `is_bst_satisfied` is a class method of the `BST` class. You cannot see the rest of the code as it is hidden. As `is_bst_satisfied` is a class method, please make sure that you don't change the indentation of the code provided to you. You are required to write your solution under the method prototype and return `True` or `False` from the method.

Good luck!

```
def is_bst_satisfied(self):
    return self.bst_helper(self.root, 0, "")

def bst_helper(self, current, max, type):
    l_satisfied = True
    r_satisfied = True
    if current.left:
```

```
        if current.left:
            if current.left.data > current.data:
                return False
            elif type == "R" and current.left.data
                return False
            else:
                l_satisfied = self.bst_helper(curr
        if current.right:
            if current.right.data < current.data:
                return False
            elif type == "L" and current.right.dat
                return False
            else:
                r_satisfied = self.bst_helper(curr
        if not l_satisfied or not r_satisfied:
            return False
        return True
```

Show Results Show Console

0.82s

2 of 2 Tests Passed

Result	Input	Expected Output	Actual Output	Reason
✓	is_bst_satisfied(tree)	True	True	Succeeded
✓	is_bst_satisfied(tree)	False	False	Succeeded

← Back

Introduction - Insertion and Search

Next →

Solution Review: Checking the BST pr...

Mark as Completed

Report an Issue

Ask a Question (https://discuss.educative.io/tag/exercise-checking-the-bst-property__binary-search-trees__data-structures-and-algorithms-in-python) educative

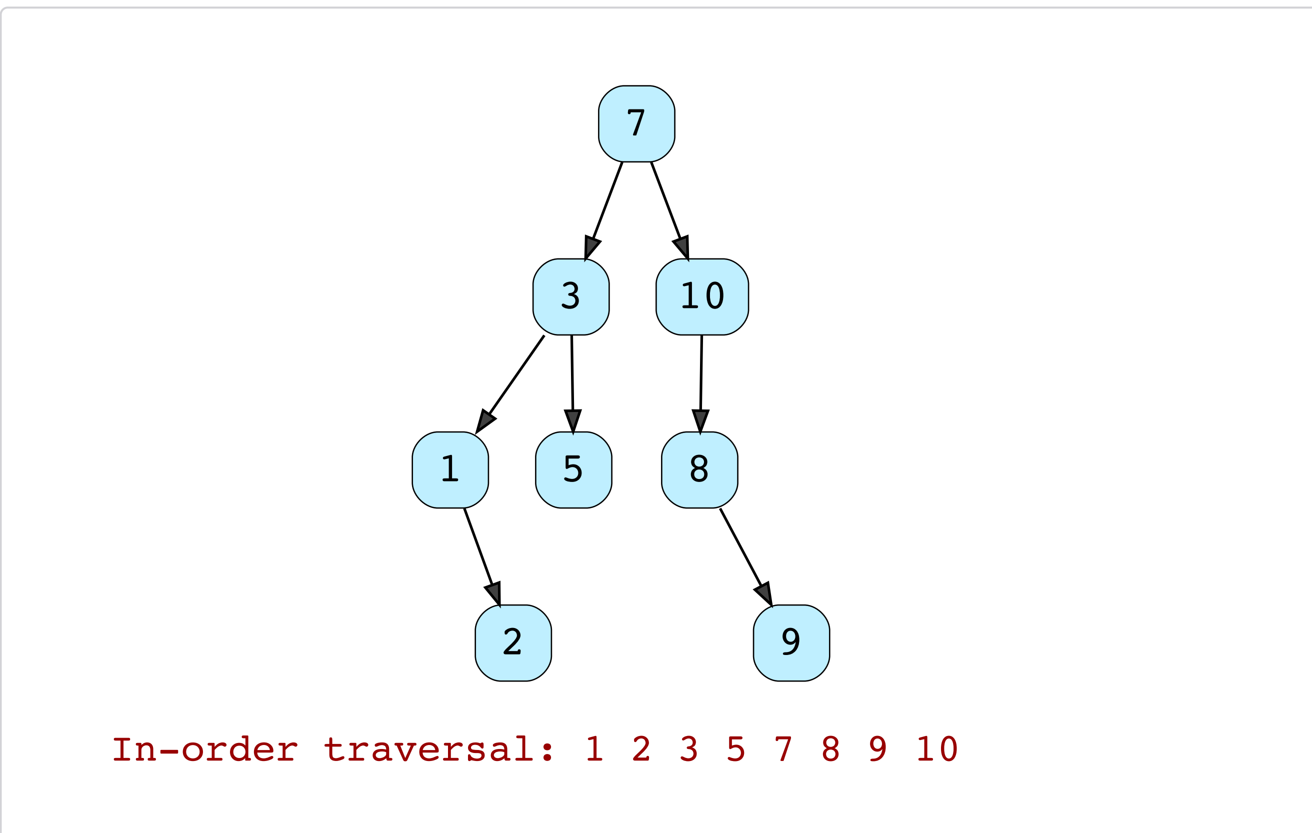
Solution Review: Checking the BST property

This lesson contains the solution review for Checking the BST property challenge.

We'll cover the following

- Implementation
- Explanation

Recall the in-order traversal that we learned in the Binary Trees chapter. The in-order traversal of a Binary Search Tree gives us the list of nodes in sorted order.



In the code widget below, we have implemented the in-order traversal for `BST` and you can confirm the traversal in the illustration above.

```

    def find(self, data):
        if self.root:
            is_found = self._find(data, self.root)
            if is_found:
                return True
            return False
        else:
            return None

    def _find(self, data, cur_node):
        if data > cur_node.data and cur_node.righ
            return self._find(data, cur_node.righ
        elif data < cur_node.data and cur_node.le
            return self._find(data, cur_node.left)
        if data == cur_node.data:
            return True

bst = BST(7)
bst.insert(3)
bst.insert(10)
bst.insert(5)
bst.insert(1)
bst.insert(8)
bst.insert(9)
bst.insert(2)

bst.inorder_print_tree()
```

Output 0.73s

```
1
2
3
5
7
```

We have discussed in-order traversal above because we'll be using a similar idea to check whether a tree satisfies the BST property or not. If we traverse a binary tree in-order and it results in a sorted list, then the tree satisfies the BST property.

Implementation

Now let's discuss the implementation of the solution we provided for the challenge in the previous lesson.

```
1  def is_bst_satisfied(self):
2      def helper(node, lower=float('-inf'), upper=f
3          if not node:
4              return True
5
6          val = node.data
7          if val <= lower or val >= upper:
8              return False
9
10         if not helper(node.right, val, upper):
11             return False
12         if not helper(node.left, lower, val):
13             return False
14         return True
15
16     return helper(self.root)
```

Explanation

In the `is_bst_satisfied` method, we define an inner method on **line 2**, `helper`, which takes `node`, `lower` and `upper` as input parameters. On **line 3**, we have the base case which caters to an empty tree or a `None` node. If `node` is `None`, `True` is returned from the method on **line 4**. Otherwise, the execution proceeds to **line 6** where `val` is made equal to `node.data`.

Next, we check if `val` is less or equal to `lower` or if `val` is greater or equal to `upper` on **line** 7. If any of the two conditions is `True`, `False` is returned from the method on **line 8**. This is because the value of the current node should be greater than all the values of the children in the left subtree, and it should be less than all the values of the children in the right subtree.

Now that we have checked the BST property for the current node, it's time to check it for the subtrees. On **line 10**, we make a recursive call to the right subtree of the current node. `node.right` is passed as `node`, `val` is passed as `lower` while `upper` stays the same. `lower` is now the lower bound for the right subtree as all the children in the right subtree have to be greater than the value of the current node. If the recursive call returns `False`, the condition on **line 10** will evaluate to `True` and `False` will be returned from the method.

Similarly, the left subtree is evaluated through a recursive call on **line 12**. Now `val` is passed as `upper` for the recursive call as all the children in the left subtree have to be less than the value of the current node.

If none of the conditions before **line 14** evaluate to `True`, `True` is returned on **line 14** declaring that the BST property is satisfied.

You can run the following code where we have the entire implementation of the `BST` class that we discussed in this chapter.

```
    def inorder_print_tree(self):
        if self.root:
            self._inorder_print_tree(self.root)

    def _inorder_print_tree(self, cur_node):
        if cur_node:
            self._inorder_print_tree(cur_node.left
            print(str(cur_node.data))
            self._inorder_print_tree(cur_node.rigl

```

```
    def find(self, data):
        if self.root:
            is_found = self._find(data, self.root)
            if is_found:
                return True
            return False
        else:
            return None

    def _find(self, data, cur_node):
        if data > cur_node.data and cur_node.right
            return self._find(data, cur_node.right
        elif data < cur_node.data and cur_node.le
            return self._find(data, cur_node.left)
        if data == cur_node.data:
            return True

    def is_bst_satisfied(self):
```

Output 0.85s

```
True
False
```

Congratulations! We have completed the Data Structures part of the course. Now, we'll focus on algorithms in the remaining course. I hope you were able to enjoy learning about data structures. Happy learning!

← Back

Exercise: Checking the BST property

Next →

Quiz

Mark as Completed

Report an Issue

Ask a Question (https://discuss.educative.io/tag/solution-review-checking-the-bst-property__binary-search-trees__data-structures-and-algorithms-in-python)