binary_search.py

```
'''
Author   :    Hameed Abdul
Date     :    3/16/18
Purpose  :    Implement a personal interpertation of
Binary Search from lecture
'''

from heap_sort import heap_sort
import numpy as np

class Node:
    '''
    A single element in the Binary Tree. Each node can have at most two children
    '''

    def __init__(self, value, left=None, right=None):
        self.value = value

        if self.is_greater_than(left):
            self.left = left
        else:
            self.left = None

        if self.is_less_than(right):
            self.right = right
        else:
            self.right = None

    def is_greater_than(self, left_val):
        '''
        Checks if input value is less than the current node's value.
        Returns True/False

        :param left_val: Value to set a Left Child
        :return: Boolean Value
        '''
        if left_val != None:
            if left_val < self.value:
                return True
        return False

    def is_less_than(self, right_val):
        '''
        Checks if input value is greater than the current node's value.
        Returns True/False
        :param right_val: Value to set a Right child
        :return: Boolean Value
        '''
        if right_val != None:
            if right_val > self.value:
                return True
        return False

    def set_right(self, right):
        '''
               Sets the right child of a Node IF the input Node's value is greater than the current Node's value
               :param right: Value to set as left child
               :return:
       '''
        if self.is_less_than(right.value):
            self.right = right

    def set_left(self, left):
        '''
        Sets the left child of a Node IF the input Node's value is less than the current Node's value
        :param left: Value to set as left child
        :return:
        '''
        if self.is_greater_than(left.value):
            self.left = left

    def get_right_value(self):
        return self.right.value
    def get_left_value(self):
        return self.left.value

    def get_value(self):
        return self.value


class BinarySearchTree:
    '''
    A DataStructure, that resembles a tree. Binary Trees have the constrain each node to
    having only two children. Where the Right Child is greater than the parent and
    the Left Child is less than the parent. With the maintenance of a balanced Binary Tree,
    These constraint allow us to quickly query the tree with efficient searches.

    '''

    def __init__(self, root=None):
        """
        Constructor with the option of adding a node to the root or not
        :type root: Node object
        """
        if root != None:
            self.root = root
        else:
            self.root = None

    def append(self, new_val):
        """
               Walks through the binary tree, creates a new Node with the new value and appends that Node to the tree
               Accordingly.

               :param new_val: int Val to be added to tree
               :return: Binary Tree with new value
       """
        if self.root is None:
            self.root = Node(new_val)
        else:
            self._append(self.root, new_val)

    def _append(self, curr_node, new_value):

        # Find a spot, compare and set accordingly
        if new_value < curr_node.value:
            if curr_node.left is not None:
                # Traverse left subtree until you find a spot
                self._append(curr_node.left, new_value)
            else:  # Found empty spot on left subtree
                curr_node.left = Node(new_value)
        elif curr_node.value < new_value:
            if curr_node.right is not None:
                # Traverse right subtree until you find a spot
                self._append(curr_node.right, new_value)
            else:  # Found empty spot on right subtree
                curr_node.right = Node(new_value)

    def print_in_order(self, root):
        '''
        Traverses tree and prints smallest to largest value in order of Left Child - Root - Right Child
        :return:
        '''
        if root != None:
            self.print_in_order(root.left)
            print(root.value)
            self.print_in_order(root.right)

    def search_for(self, curr_node, value):
        '''
        Search the tree for a specified queried value.
        :param value: Query value to search for in the tree
        :return: Inform user of location or absence of node in the tree
        '''

        # If the tree is empty or not in the list
        if curr_node is None:
            print(value, " is not in the tree")

        # If the Value is found
```

```python
            elif curr_node.value == value:
                print(value, " is in the tree")

            else:
                if value < curr_node.value:
                    self.search_for(curr_node.left, value)
                elif curr_node.value < value:
                    self.search_for(curr_node.right, value)

    def array_to_binarytree(self, arr):
        '''
        Use to converted array to Binary Search Tree.
        :param arr: Array of integer elements to pass in
        :return: Node and all right and left (sub)trees
        '''
        # Append only when there is a value assigned and within index of the array
        if len(arr) == 0:
            return None

        # Heap sort input array
        heap_sort(arr)

        # Size, Middle value, and appropriate nodes from current input array
        size = len(arr)
        middle = size // 2
        new_node = Node(arr[middle])

        # Recursively call to append all Left and Right (sub) children
        new_node.left = self.array_to_binarytree(arr[:middle])
        new_node.right = self.array_to_binarytree(arr[middle + 1:])

        # Set the tree's root to arr[middle]
        # Return the current node and it's children
        self.root = new_node
        return new_node

    def delete_all(self):
        '''
        Delete all nodes from tree
        :return: Empty Tree
        '''
        self.root = None
        print("Deleted all nodes from tree")

def main():
    x = Node(1)

    y = Node(2)
    x.set_right(y)

    w = Node(0)
    x.set_left(w)
    z = Node(22)
    # print(x.get_value(), x.get_left_value(), x.get_right_value())

    bt = BinarySearchTree()
    bt.append(22)
    bt.append(222)
    bt.append(1)
    bt.append(-2)

    bt.print_in_order(bt.root)
    bt.search_for(bt.root, 1)
# main()

def test_cases():
    # Random Arrays to test
    rand_arr1 = np.random.randint(256, size=8)
    rand_arr2 = np.random.randint(256, size=8)

    # Given Test Cases
    sorted_arr = [0, 1, 2, 3, 4, 5, 7, 9, 10, 12, 14, 18, 20]
    unsorted_arr = [3, 0, 1, 10, 18, 4, 7, 20, 15, 9, 2, 12, 14, 5]

    # Create, Fill, Search and Empty from each test case
    bt = BinarySearchTree()
    bt.array_to_binarytree(rand_arr1)
```

```
        print('Random Numpy Array: ', rand_arr1)
        bt.print_in_order(bt.root)
        bt.search_for(bt.root, 11)
        bt.delete_all()

        bt.array_to_binarytree(rand_arr2)
        print('Random Numpy Array #2: ', rand_arr2)
        bt.print_in_order(bt.root)
        bt.search_for(bt.root, 32)
        bt.delete_all()

        bt.array_to_binarytree(sorted_arr)
        print('Random Numpy Array', sorted_arr)
        bt.print_in_order(bt.root)
        bt.search_for(bt.root, 7)
        bt.search_for(bt.root, 244)
        bt.delete_all()

        bt.array_to_binarytree(unsorted_arr)
        print('Random Numpy Array', unsorted_arr)
        bt.print_in_order(bt.root)
        bt.search_for(bt.root, 3)
        bt.search_for(bt.root, 21)
        bt.delete_all()


    test_cases()
```