Homework #8
Due by Thursday 5/2, 11:55pm

Submission instructions:

1. In this assignment, we are focusing mostly on the **structure** of the search trees and on the order the keys are set in them legally. We are less interested here in the values associated to these keys (as defined in the *map ADT*).
   Therefore, in these cases, associate `None` as the value for all keys.

2. In this assignment, we provided 'BinarySearchTreeMap.py' file (with the implementation of a Map ADT, using a binary search tree), as implemented in class.
   Note: You are not allowed to make any changes to this file. You should only use it.

3. For this assignment, you should turn in five files:
   a. A '.pdf' file containing your answers to questions 1and 2c.
      Name this file 'YourNetID_hw8.pfd'.

   b. 3 '.py' files: each one includes the functions you wrote for questions 2, 3 and 4.
      Name your files: 'YourNetID_hw8_q2.py', 'YourNetID_hw8_q3.py', and 'YourNetID_hw8_q4.py'.
      Note: your netID follows an abc123 pattern, not N12345678.

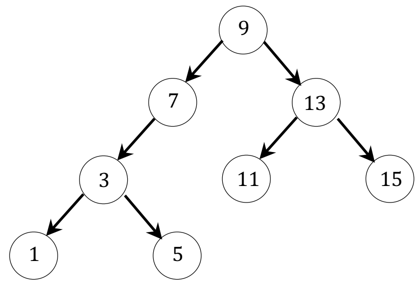
   c. A '.py' file containing the definition of the `BinarySearchTreeMap` class, including the additional method implemented in question 5 of this assignment.
      Name this file 'YourNetID_BinarySearchTreeMap.py'.
      Note: Besides this question, all other implementations, should work with the 'BinarySearchTreeMap.py' file that we provided with this assignment.

4. You should submit your homework via Gradescope. For Gradescope's autograding feature to work:
   a. Name all functions and methods exactly as they are in the assignment specifications.

   b. Make sure there are no print statements in your code. If you have tester code, please put it in a "main" function and do not call it.

Question 1:

Given the following binary search tree `bst`:



We are executing the following sequence of operations (one after the other):

```
bst[6] = None
bst[12] = None
bst[4] = None
bst[14] = None
del bst[7]
del bst[9]
del bst[13]
del bst[1]
del bst[3]
```

Draw the resulting tree after each one of the operations above.

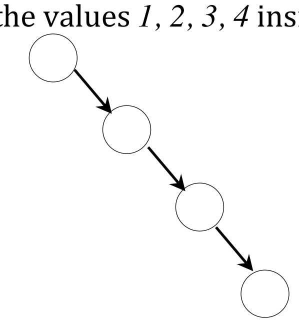
Question 2:

a. Implement the following function:

```
def create_chain_bst(n)
```

This function gets a positive integer `n`, and returns a binary search tree with `n` nodes containing the keys *1*, *2*, *3*, …, *n*. The structure of the tree should be one long chain of nodes leaning to the right.

For example, the call `create_chain_bst(4)` should create a tree of the following structure (with the values *1*, *2*, *3*, *4* inside its nodes in a valid order):



Implementation requirement: In order to create the desired tree, your function has to construct an empty binary search tree, and can then only make repeated calls to the insert method, to add entries to this tree.

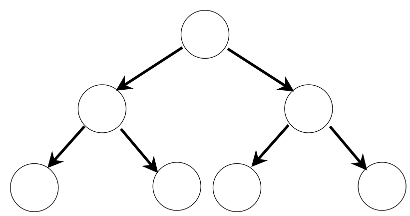
b. In this section, you will show an implementation of the following function:

```
def create_complete_bst(n)
```

`create_complete_bst` gets a positive integer `n`, where `n` is of the form $n=2^k-1$ for some non-negative integer k.
When called it returns a **binary search tree** with `n` nodes, containing the keys $1, 2, 3, \ldots, n$, structured as a **complete** binary tree.

Note: The number of nodes in a complete binary tree is 2^k-1, for some non-negative integer k.

For example, the call `create_complete_bst(7)` should create a tree of the following structure (with the values $1, 2, 3, 4, 5, 6, 7$ inside its nodes in a valid order):



You are given the implementation of `create_complete_bst`:

```
def create_complete_bst(n):
    bst = BinarySearchTreeMap()
    add_items(bst, 1, n)
    return bst
```

You should implement the function:

```
def add_items(bst, low, high)
```

This function is given a binary search tree `bst`, and two positive integers `low` and `high` (`low` $\leq$ `high`).
When called, it adds all the integers in the range `low` ... `high` into `bst`.

Note: Assume that when the function is called, none of the integers in the range `low` ... `high` are already in `bst`.

Hints:

- Before coding, try to draw the binary search trees (structure and entries) that `create_complete_bst(n)` creates for `n=7` and `n=15`.
- It would be easier to define `add_items` recursively.

c. Analyze the runtime of the functions you implemented in sections (a) and (b)

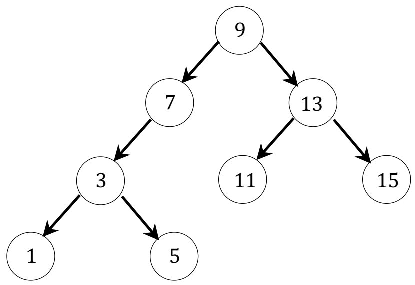
Question 3:

Implement the following function:

```
def restore_bst(prefix_lst)
```

The function is given a list `prefix_lst`, which contains keys, given in an order that resulted from a prefix traversal of a **binary search tree**.
When called, it creates and returns the binary search tree that when scanned in prefix order, it would give `prefix_lst`.

For example, the call `restore_bst([9, 7, 3, 1, 5, 13, 11, 15])`, should create and return the following tree:



Notes:
1. The runtime of this function should be **linear**.
2. Assume that `prefix_lst` contains integers.
3. Assume that there are no duplicate values in `prefix_lst`.
4. You may want to define a helper function.

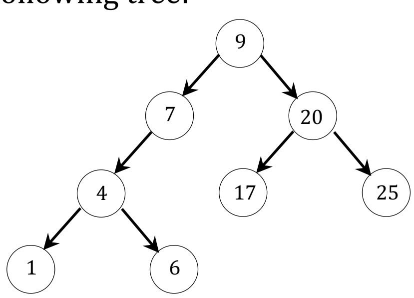
Question 4:

Implement the following function:

```
def find_min_abs_difference(bst)
```

The function is given a binary search tree `bst`, where all its keys are non-negative numbers.
When called, it returns the **minimum absolute difference** between keys of any two nodes in `bst`.

For example, if `bst` is the following tree:



The call `find_min_abs_difference(bst)` should return 1 (since the absolute difference between 6 and 7 is 1, and there are no other keys that their absolute difference is less than 1).

<u>Implementation requirement</u>: The runtime of this function should be **linear**. That is, if `bst` contains n nodes, this function should run in $\Theta(n)$.

<u>Hint</u>: To meet the runtime requirement, you may want to define an additional, recursive, helper function, that returns more than one value (multiple return values would be collected as a tuple).

<u>Question 5:</u>

Modify the implementation of the `BinarySearchTreeMap` class, so in addition to all the functionality it already allows, it will also support the following method:

```
def get_ith_smallest(self, i)
```

This method should support indexing. That is, when called on a binary search tree, it will return the `i`-th smallest key in the tree (for `i=1` it should return the smallest key, for `i=2` it should return the second smallest key, etc.).

For example, your implementation should behave as follows:

```
>>> bst = BinarySearchTreeMap()
>>> bst[7] = None
>>> bst[5] = None
>>> bst[1] = None
>>> bst[14] = None
>>> bst[10] = None
>>> bst[3] = None
>>> bst[9] = None
>>> bst[13] = None
>>> bst.get_ith_smallest(3)
5
>>> bst.get_ith_smallest(6)
10
>>> del bst[14]
>>> del bst[5]
>>> bst.get_ith_smallest(3)
7
>>> bst.get_ith_smallest(6)
13
```

<u>Implementation requirements</u>:

1. The runtime of the existing operations should remain as before (worst case of $\Theta(height)$). The runtime of the `get_ith_smallest` method should also be worst case of $\Theta(height)$.
2. You should raise an `IndexError` exception in case `i` is out of range.

<u>Hints</u>:

1. You may want to add attributes to the Node objects to help you search for the i^{th} smallest element. To keep them updated, it could require you to modify the insert and delete methods as well.
2. You may want to define additional helper methods.