Project: Binary Search Trees

A *binary search* tree is a method to organize data, together with operations on these data (i.e., it is a *data structure).* In particular, the operation that this organization wants to perform really fast is *searching.* The data are records that may contain many fields (recall the student record from Chapter 11), but one field is specifically used for ordering the data records; this field is called the record *key.* In order to simplify things, from now on we will assume that the data record contains only this key, and the keys are integers; but we don't require that keys are unique (so, the same key may show up multiple times). Then a binary search tree consists of nodes that are of the following form:

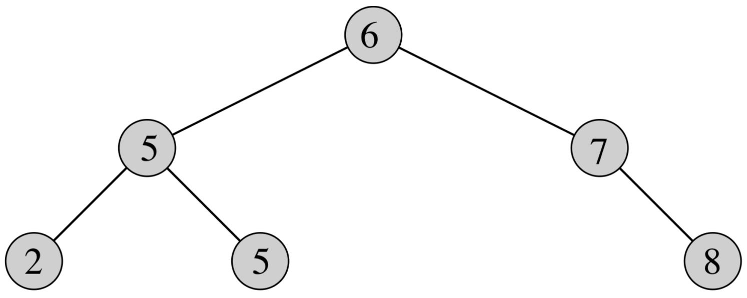
[key, Left subtree, Right subtree]

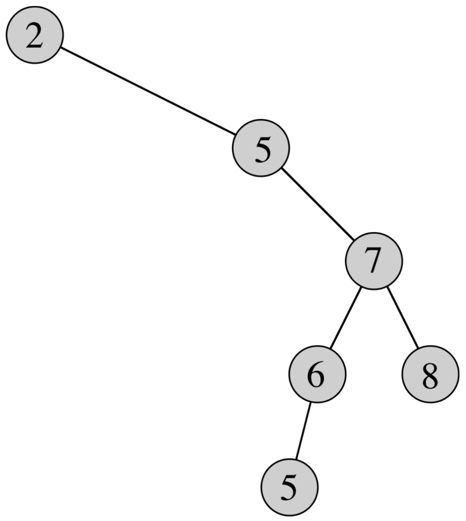
Notice that a node is itself a record. It is a list with three elements: the first element is an integer (key), and the next two elements *are themselves binary search trees,* a Left subtree and a Right subtree! So the definition of a binary search tree is a *recursive* one. Before we define binary search trees though, it is simpler to first define *binary trees* (without the "search" component):

Definition (binary tree): A binary tree T is

- either an empty tree (T=[]) (base case)
- or T=[key, L, R], where L, R are binary trees (called the *subtrees* of T).

We can draw a binary tree using "nodes" to hold the keys and lines (*edges*) that show what are the left and right subtrees of the "nodes". Here is such a conceptual drawing of two binary trees[1]:


(a)


(b)

The actual Python picture of the tree (a), following the definition, is:

T = [6, [5, [2, [], []], [5, [], []]], [7, [], [8, [], []]]]

This looks confusing, but it isn't hard to see what's going on. In the conceptual picture, you can see that there are three "nodes" with no subtrees whatsoever (such "nodes" are called *leaves*); they are the

1 All figures and pseudo-code in this project come from Chapter 12 of "Introduction to Algorithms", 3rd ed., by T. Cormen, C. Leiserson, R. Rivest, C. Stein. MIT Press 2009.

nodes with key values 2,5, and 8. These nodes are themselves binary trees, but with empty Left and Right subtrees, so they should look like [2, [], []], [5, [], []], [8, [], []]. Let's name them

- T1 = [2, [], []]
- T2 = [5, [], []]
- T3 = [8, [], []]

Moving a level up, note that T1, T2 are the Left and Right subtrees of the "node" with key value 5; so this "node" is (according to the definition) [5, T1, T2]. Let's name it

- T4 = [5, T1, T2]

Also T3 is the Right subtree of "node" with key value 7, and this "node" has no Left subtree; therefore, it is [7, [], T3]. Let's name it

- T5 = [7, [], T3]

Now, notice that T4, T5 are the Left and Right subtrees of the "node" with key value 6 (since this is the top-most "node", i.e., the node where our recursive definition started from, it is called the *root* of the whole binary tree). So this node should look like [6, T4, T5]. But this is exactly what T is! (tarting from [6, T4, T5], start replacing the names with their actual list values and you'll get T)

> **Practice problem:** Repeat the same process for tree (b) in the figure, to find out how it really looks in Python, using the definition.

A binary *search* tree is itself a binary tree, but we need to add to the definition one property that this tree must satisfy; namely, we will require that in a binary search tree T = [k, L, R] the key values in the Left subtree L are all *smaller or equal* to key value k, and the key values in the Right subtree R are all *greater or equal* to k. So, the new definition is:

Definition (binary search tree): A binary search tree T is

- either an empty tree (T=[]) (base case)
- or T=[key, L, R], where L, R are binary search trees (called the *subtrees* of T) with the following property: the key values in L are all *smaller or equal* to key, and the key values in R are all *greater or equal* to key.

Note that because we now require L, R to be themselves binary *search* trees (instead of simply binary trees as before), the key values inside L and R also have this nice key property. Trees (a) and (b) above are actually binary search trees, because their keys satisfy this extra property.

So, what kind of operations does this data structure support? Here are some of them:

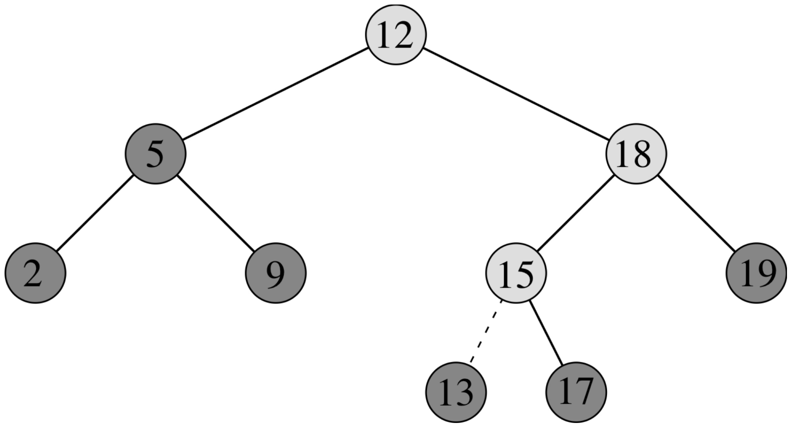
Searching: Now, it should be clear why we would like to organize our data into a binary search tree: Suppose we want to search for a record with key value x in a binary search tree T. If T=[] (empty), then we return -1 or "didn't find it" or "x doesn't exist" or something along these lines, that indicates that a record with key value x doesn't exist in our data. If T is not empty, then it must be of the form T=[k,L,R]; in this case, if x=k then this is our record, and we return its data (in our case we don't have any more data to return, so probably we want to return something like 0, or "I found it!" or something along these lines). But if x is not equal to k, then it may be in L or R; but which one? Here is exactly the point where we use the nice key values property: if x<k, then we should search in L, and if x>k, then we should search in R. The point is that once you determine which subtree to search, then you don't have to even bother with the other subtree.

Finding minimum/maximum: It is also extremely easy to find the record with minimum key value:

all you have to do is go “left” all the way, until you cannot go anymore “left” (why does this work?). Finding the maximum key value is symmetric (I'll let you think about it yourselves).

Sorting: By now it should be obvious how to output all key values (remember! we allow multiple instances of the same key value!) sorted from smallest to largest.

Insertion: Recall that we initially start with an empty data structure (T=[]); then we build it up by *inserting* new records (each with its own key) one-by-one. In order to insert a record with key x, you need to first find exactly where x should be inserted in order that the new tree is still a binary search tree *(hint: you already know how to do this!)*. An example of insertion is the following figure:

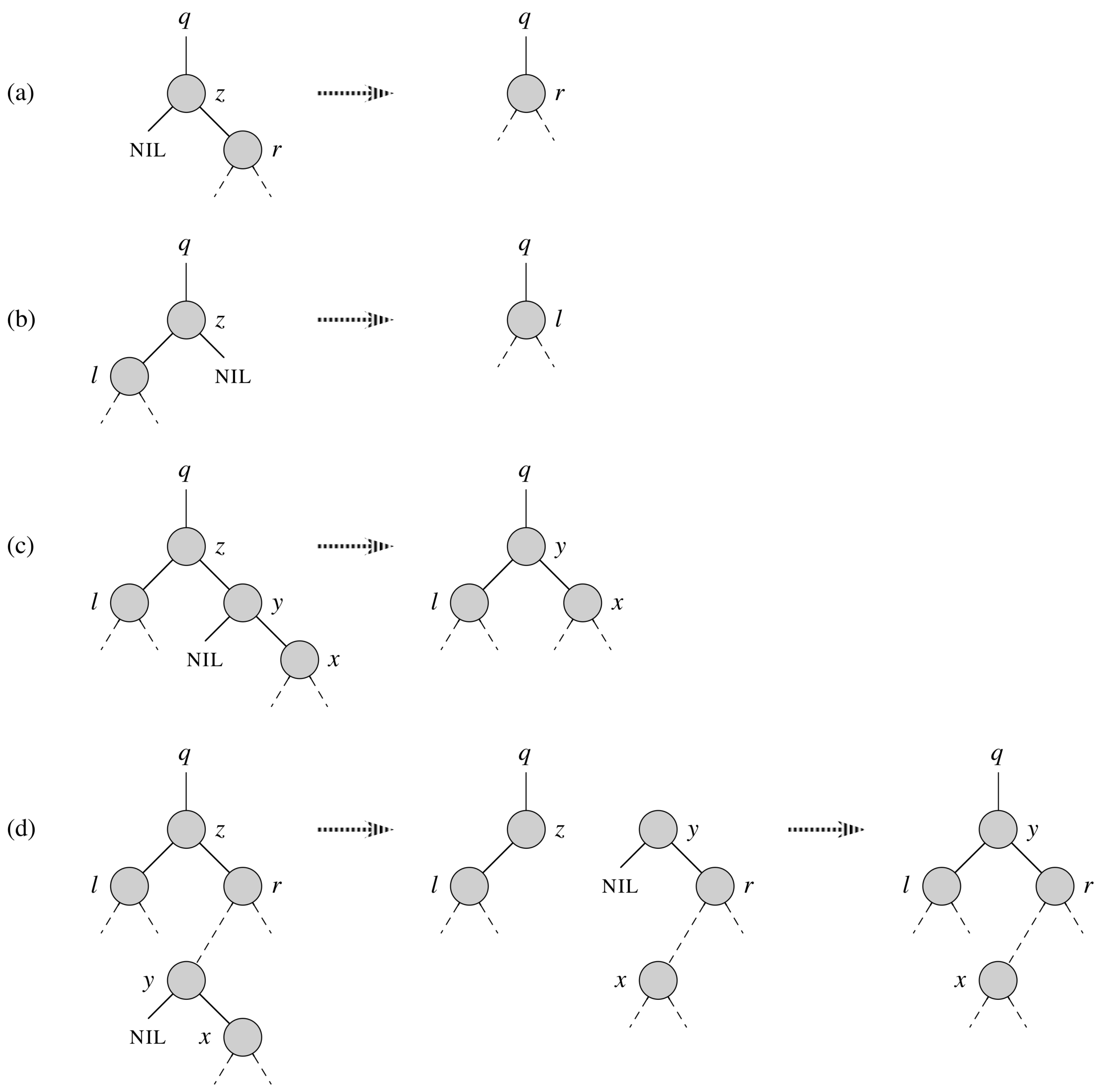


In this figure, key 13 was inserted in the tree; the light-grey nodes show the path of nodes that the insertion function “looked” before finding the one (with key 15) that got 13 as a new left subtree.

Deletion: Deletion is the most complicated operation to perform in binary search trees. It is probably best described by the following figure, that describes what happens when a node z=[z.key, l, r] (z is possibly the left of right subtree of a node q, or is the root of the whole tree itself) is deleted. There are four distinct cases, with case (d) being the complicated one: first you “split” the “node” z into two parts (trees), then you find the minimum[2] key of r (let's say it's y), you “promote” it to be the new root of r as shown, and then you combine again the two trees into one, as shown.[3] Those of you that are observant have already noticed that case (c) is, in fact, the special case of (d) when y is the same as r, i.e., y/r is already the minimum key node for the right subtree of z.

Practice problem: Delete 18, 12, 15 successively from the tree above (in this order).

2 We already know how to do this!
3 In the figure, NIL stands for an empty tree [].



Project

Implement a class called binary_search_tree that contains at least the following internal variables and methods (in the process of designing your class, you may end up implementing/using more variables/methods; it's totally up to you):

- T: the variable that contains the tree
- __init__: the class constructor; must set the initial tree to 'empty' (T=[])
- getTree: method that returns the tree T
- searchTree: takes a key value x, and returns -1 if x is not found in T, and 0 otherwise.
- minimumTree, maximumTree: return the minimum, maximum keys of T, respectively

- increasing_orderTree: outputs a sorted list of all the keys in T (duplicates are allowed)
- insertTree: takes a key value x, and inserts it in T
- deleteTree: takes a key value x, and deletes it from T (or does nothing if x isn't in T)

Then, use an object from your class to write a sorting program. The user is asked to give integer values one after the other, separated by a space, and then the program outputs the same numbers but sorted from smallest to largest, and then the same numbers sorted from largest to smallest (*hint: you may want to implement a method decreasing_orderTree, doing the symmetric job of increasing_orderTree).*