National University of Singapore
School of Computing
CS1010S: Programming Methodology
Semester I, 2019/2020

Mission 10
What Sort Of Sorcery Is This?

Release date: 11 October 2019
Due: 20 October 2019, 23:59

Required Files

- mission10-template.py

Background

"Phew. After all that work, I'll never forget how to write a sorting spell again."

You look wistfully at your neat wardrobe knowing it'll be a mess again in no time.

"How fine it would be if I could enchant the wardrobe so that it sorts itself out. Then, I'd just have to toss clothing in and they would end up in the right spot, neatly folded. Ahhh! All the time that would be saved from casting sorting spells and from hunting for particular pieces of clothing..."

Your scheming mind begins to churn.

Administrivia

The template file `mission10-template.py` has been provided for you to write your answers in. It also includes sample input and output. In addition to these samples, you should test your functions with your own test cases.

Please use IDLE or another Integrated Development Environment (IDE) to write and test your code before you copy it onto Coursemology. You will be able to run a set of test cases to verify your the correctness of your answers on Coursemology, but only for limited number of times. Click "Finalize Submission" when you are done and your graders will assess your work in due course.

In this mission, you will implement a **binary search tree** and use it to do some sorting. So as not to undermine your learning, you are barred from using Python built-in sorting functions like `sorted` and `.sort` in this mission.

As usual, do not break the abstraction layer.

This mission consists of **three** tasks.

The Binary Search Tree

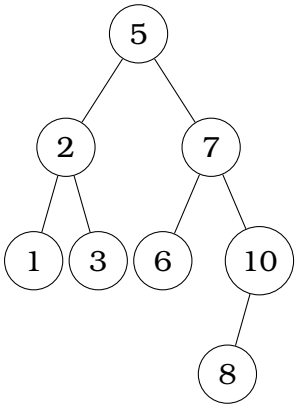
In this mission, we will be manipulating a binary tree abstract data type. This data type will be the "wardrobe" that we will try to enchant.

Recall that a binary tree is a data structure in which each node has at most two child nodes. In this mission, we shall refer to a node's value as the node *entry*. The node's left and right children as left and right branches.

The tree adheres to the following rules:

- The left branch of a node contains only nodes with entries less than or equal to the node's entry.
- The right branch of a node contains only nodes with entries greater than the node's entry.

Here is an example of a binary tree:

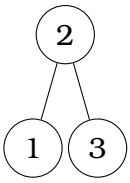


The nodes in the left branch of the node containing 5 are less than 5 and the numbers on the right branch of it are more than 5. The two rules similarly apply for the other nodes.

After we have enchanted it, the "wardrobe" will automatically sort the items for us, i.e. at any point in time, the tree contains a sorted sequence. To visit the node entries in the correct order, follow the following set of rules:

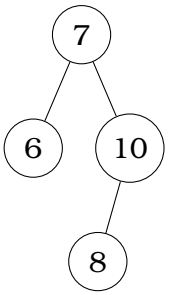
1. Visit the left branch. If the left branch contains other branches, apply this set of rules on it.
2. Then visit the node's entry.
3. Lastly, visit the right branch. If the right branch contains other branches, apply this set of rules on it.

An example is in order. Let's look at the left branch of the node containing 5.



Let's apply the visiting rule on this subtree. We visit the node's left branch, whose sole entry is 1, then visiting the node's own entry will give us 2. Lastly, after visiting the right branch, we get the entry 3. After applying this rule, the list of entries we would have

visited (in order) is [1, 2, 3]. Let's look at another example. The right branch of 5 is given below.



When we apply on this tree, we will visit 6, 7, then we will visit the right branch of 7 where we have additional subtrees in the node. Therefore, we have to apply the same rule on it recursively and we will get 8 and then 10. When we collect the entries we have visited, we get [6, 7, 8, 10].

Now we apply the rule to the whole of the tree, which rooted at the node with entry 5. On the visit to the left branch, we visit the entries [1, 2, 3]. Next, we visit the entry 5. Finally, we visit the entries on the right branch. Thus, we get [1, 2, 3, 5, 6, 7, 8, 10], which is a sorted list in ascending order.

Task 1: Binary Tree ADT (12 marks)

Before we enchant the wardrobe to sort whatever gets thrown into it, we have to (re)orgainse the shelving a little. In this task, we will prepare the binary tree abstract data type.

(a) Define a function `build_tree` that accepts as input: the entry, the left branch and the right branch. The function should return a tree abstract data type. You are to decide how to implement the ADT. **(2 marks)**

(b) Define accessor functions `entry`, `left_branch` and `right_branch` for the tree abstract data type. These functions should work for non-empty trees. **(6 marks)**

(c) Define a function `make_empty_tree` that returns a representation of an empty binary tree. When a tree is not empty, it should contain an entry and two subtrees. When it is empty, it should contain nothing. **(2 marks)**

(d) Write a function `is_empty_tree` that takes in a tree data type and returns `True` if the tree is empty, `False` otherwise. **(2 marks)**

Note: Once you are done with **Task 1**, use the predefined `print_tree` to test your implementation. The function will print a command-line-ish spartanic graphical representation of your tree.

Task 2: Enchant it! (15 marks)

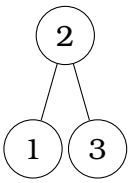
In this task, we will enchant the wardrobe according to the enchantment spelt out above.

The magic in this enchantment happens in the `insert_tree` function, which inserts an item into a tree. This function will ensure that items inserted into the tree remains "enchanted", i.e. sorted. It ensures that the items get inserted into the correct branch of every node traversed. If an item is less than or equal to the node's entry, the item should be inserted into the left branch. If an item is more than the node's entry, it should go into the right branch.

Therefore, inserting the item **`x`** into a tree has three cases:

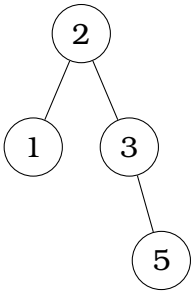
- If the tree that we are inserting into is empty, we return a tree object with **`x`** as the node's entry and empty tree as its subtrees.
- If the tree is non-empty, we have two more cases:
  - If **`x`** is less than or equal to the node's entry, we will return a new tree with **`x`** inserted into the left branch.
  - If **`x`** is more than the node's entry, we will return a new tree with **`x`** inserted into the right branch.

Say we want to insert 5 into this tree:



Since 5 is more than 2, we insert 5 into the right branch. The right branch's entry contains 3, which is less than 5, therefore we insert 5 into the right branch. Since the right branch of the node containing 3 is an empty tree, we simply make a new tree containing 5.

This is what we will get after inserting 5 into the tree:



Tasks

(a) Define a function `insert_tree` that takes in the item to insert and the tree to insert into and returns the tree with the newly inserted item. The function should insert item according to the algorithm as described above. **(4 marks)**

```
>>> t1 = build_tree(2, build_tree(1, make_empty_tree(),
                                     make_empty_tree()),
                       build_tree(3, make_empty_tree(),
                                     make_empty_tree()))
>>> print_tree(insert_tree(5, t1))
  2
1   3
     5
>>> t2 = build_tree(5, build_tree(2, build_tree(1, make_empty_tree(),
                                                    make_empty_tree()),
                                     make_empty_tree()),
                       build_tree(7, make_empty_tree(),
                                     build_tree(10, make_empty_tree(),
                                                    make_empty_tree())))
>>> print_tree(insert_tree(6, t2))
   5
 2   7
1   6 10
```

(b) What is the time complexity of `insert_tree`? **(1 mark)**

Recap: List concatenation and slicing are each $O(n)$ in time complexity.

(c) Define a function `contains` that searches for an item in the tree. It should take in a value and a tree and returns True if the value is in the tree, or otherwise, False. **(4 marks)**

(d) What is the time complexity of `contains`? **(1 mark)**

(e) Write a function `flatten` that takes in a tree and returns a list that visits the nodes in the tree according to the rule described in the section "The Binary Search Tree". If the tree is empty, `flatten` should return an empty list. **(4 marks)**

Hint: The list concatenation operator might be helpful to you. Use `+` to concatenate multiple lists. e.g `[my_val] + some_list + [another_val] + some_other_list`

```
>>> t2 = build_tree(5, build_tree(2, build_tree(1, make_empty_tree(),
                                                    make_empty_tree()),
                                     make_empty_tree()),
                       build_tree(7, make_empty_tree(),
                                     build_tree(10, make_empty_tree(),
                                                    make_empty_tree())))
>>> print(flatten(t2))
[1, 2, 5, 7, 10]
```

(f) What is the time complexity of `flatten`? **(1 mark)**

Congratulations, your wardrobe is now enchanted, let's use it to sort some items!

Task 3: Sort it! (5 marks)

(a) Using the functions `insert_tree` and `flatten`, define a function `sort_it` that takes in a list of numbers and returns a sorted list in an ascending order. **(4 marks)**

```
>>> print(sort_it([5, 3, 2, 1, 4, 6, 7, 9]))
[1, 2, 3, 4, 5, 6, 7, 9]
>>> print(sort_it([5, 3, 2, 1, 4, -1, 6, 0, 7, 9]))
[-1, 0, 1, 2, 3, 4, 5, 6, 7, 9]
```

Hint: `accumulate` has been provided to aid you in this task.

(b) What is the time complexity of `sort_it`? **(1 mark)**