National University of Sciences & Technology (NUST)
School of Natural Sciences (SNS)
Department of Mathematics

CS250: Data Structures and Algorithm

Class: BS2k21_Mathematics – Fall 2023

Lab 12:

Non-Linear Data Structures in Python

Implementation of Binary trees and binary search trees (BSTs)

Date: 5 December, 2023

Time: 10:00 am - 01:00 pm

Instructor: Fauzia Ehsan

Submission Deadline: See LMS

National University of Sciences & Technology (NUST)
School of Natural Sciences (SNS)
Department of Mathematics

Lab 12: Implementation of Tree Data structures in Python

Introduction:

In this lab students will implement binary trees and binary search trees.

Description:

A Tree is a data structure that organizes data in memory by using nodes (also known as vertices) and edges. Nodes store the data, while edges represent paths between nodes.Trees can have any number of nodes and edges.
There are different types of trees, such as ternary trees, binary search trees, etc.

A **binary tree** is a structure with two properties: a shape property and a property that relates the keys of the elements in the structure. The shape property, binary tree is a structure in which each node is capable of having two successor nodes, called children. Each of the children, being nodes in the binary tree, can also have two child nodes, and so on, giving the tree its branching structure. The beginning of the tree node is called the root. The level of nodes refers to its distance from the root.

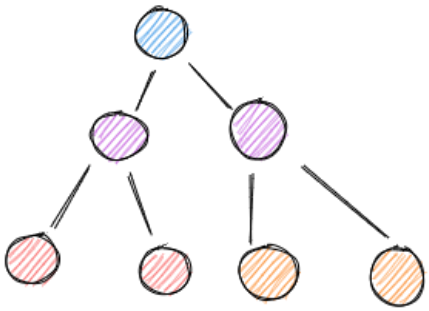
A **binary search tree** is a binary tree in which the key value in any node is greater than the key value in its left child and any of its children (the nodes in the left subtree) and less than the key value in its right child and any of its children (the nodes in the right subtree).

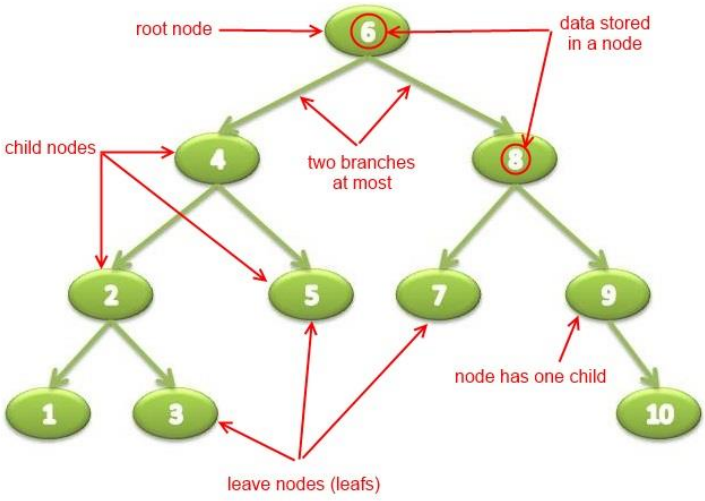
Tools/Software Requirement

Python 3/ PyCharm IDE / MS Visual Studio

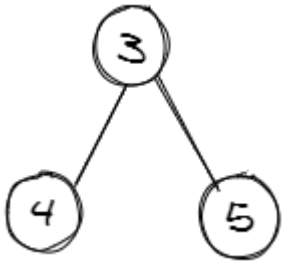
Table of Contents

Binary Tree





To begin, we'll create simple binary tree (without any of the additional properties) containing numbers as keys within nodes. Here's an example:




```
#simple class representing a node within a binary tree.
class TreeNode:
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None#create objects representing each node of the
above treenode0 = TreeNode(3)
node1 = TreeNode(4)
node2 = TreeNode(5)

#connect the nodes by setting the .left and .right properties of the
root node. And we're done! We can create a new variable tree which
simply points to the root node, and use it to access all the nodes
within the tree.node0.left = node1
node0.right = node2#Going forward, we'll use the term "tree" to refer
to the root node. The term "node" can refer to any node in a tree, not
necessarily the root.
```

Traversing a Binary Tree

A *traversal* refers to the process of visiting each node of a tree exactly once. *Visiting a node* generally refers to adding the node's key to a list. There are three ways to traverse a binary tree and return the list of visited keys:

Inorder traversal

1. Traverse the left subtree recursively inorder.
2. Traverse the current node.
3. Traverse the right subtree recursively inorder.

Preorder traversal

1. Traverse the current node.
2. Traverse the left subtree recursively preorder.
3. Traverse the right subtree recursively preorder.

Postorder traversal

1. Traverse the left subtree recursively postorder.
2. Traverse the right subtree recursively postorder.
3. Traverse the current node.

```
#implementation of inorder traversal of a binary tree
def traverse_in_order(node):
    if node is None:
        return []
    return(traverse_in_order(node.left) +
           [node.key] +
           traverse_in_order(node.right))
```

Height and Size of Binary Tree

The *height/depth* of a binary tree is defined as the length of the longest path from its root node to a leaf.

```
#computation of height/depth of a binary tree
def tree_height(node):
    if node is None:
        return 0
    return 1 + max(tree_height(node.left), tree_height(node.right))
# computation of number of nodes in a binary tree
def tree_size(node):
    if node is None:
        return 0
    return 1 + tree_size(node.left) + tree_size(node.right)
```

A Binary Search Tree (BST)

A binary search tree or BST is a binary tree that satisfies the following conditions:

1. The left subtree of any node only contains nodes with keys less than the node's key
2. The right subtree of any node only contains nodes with keys greater than the node's key

```
#Function to check if a binary tree is a binary search tree and to find
the minimum and maximum key in a binary tree
def is_bst(node):
    if node is None:
        return True, None, None

    is_bst_l, min_l, max_l = is_bst(node.left)
    is_bst_r, min_r, max_r = is_bst(node.right)
```

```
    is_bst_node = (is_bst_l and is_bst_r and
                   (max_l is None or node.key > max_l) and
                   (min_r is None or node.key < min_r))

    min_key = min(remove_none([min_l, node.key, min_r]))
    max_key = max(remove_none([max_l, node.key, max_r]))

    print(node.key, min_key, max_key, is_bst_node)

    return is_bst_node, min_key, max_key
```

Insertion in BST

Using the BST-property to perform insertion efficiently:

1. Starting from the root node, compare the key to be inserted with the current node's key
2. If the key is smaller, recursively insert it in the left subtree (if it exists) or attach it as the left child if no left subtree exists.
3. If the key is larger, recursively insert it in the right subtree (if it exists) or attach it as as the right child if no right subtree exists.

```
#Implementation of insertion
def insert(node, key, value):
    if node is None:
        node = BSTNode(key, value)
    elif key < node.key:
        node.left = insert(node.left, key, value)
        node.left.parent = node
    elif key > node.key:
        node.right = insert(node.right, key, value)
        node.right.parent = node
    return node
```

Finding a particular node in BST

```
#Implementation of finding a node
def find(node, key):
    if node is None:
        return None
    if key == node.key:
        return node
    if key < node.key:
        return find(node.left, key)
    if key > node.key: return find(node.right, key)
```

Updating value in BST

```
#Implementation of update
def update(node, key, value):
    target = find(node, key)
    if target is not None:
        target.value = value
```

Retrieving all the key-value pairs stored in BST in sorted order of keys

```
def list_all(node):
    if node is None:
        return []
    return list_all(node.left) + [(node.key, node.value)] +
list_all(node.right)
```

1.2 Applications of a Binary Search Tree

- Search applications, Syntax tree for parse expression, 3D video games, compression algorithm and etc.
  - Applications
  - more links
- Dictionary Search, Sorting and Searching informations from databases.
  - Dictionary
  - Binary Search Tree

1.3 Typical Operations of a Binary Search Tree

`insert ( newDataItem )`	**Insert data item**
`retrieve ( searchKey, searchDataItem )`	**Retrieve data item**
`remove ( deleteKey )`	**Remove data item**
`writeKeys ()`	**Output keys**
`showStructure ()`	**Output the tree structure**
`getHeight ()`	**Height of tree**
`getCount ()`	**Number of nodes in tree**

Implement a Tree Using a Python Library

As mentioned earlier, creating a tree from scratch can be time-consuming and requires writing a significant amount of code. However, there is a simpler approach to accomplish this using the `anytree` library. With the aid of the `anytree` library, you can effortlessly generate a tree without the need for extensive coding.

https://pypi.org/project/anytree/

Before utilizing the `anytree` library, it is necessary to install it using the command provided below.

```
pip install anytree
```

We are creating the tree and now we can import `Node` and `RenderTree` from the `anytree` library.

```
from anytree import Node, RenderTree
root = Node(10)
level_1_child_1 = Node(34, parent=root)
level_1_child_2 = Node(89, parent=root)
level_2_child_1 = Node(45, parent=level_1_child_1)
level_2_child_2 = Node(50, parent=level_1_child_2)
for pre, fill, node in RenderTree(root):
    print("%s%s" % (pre, node.name))

# Tree Structure
#           10
#         /    \
#        34      89
#      /    \
#     45     50
```

Determine the Output (Add screenshot)

In this case, the `Node` creates a node with two parameters: the value of the node and the parent node's name (which is optional). Since the `root` node is the only node in our tree without a parent, we will only pass the first parameter (the node's value) when creating the `root` node. The `RenderTree` method is used to display the entire tree in the output.

For Binary tree: pip install binarytrees
https://pypi.org/project/binarytree/

BST using built-in module

```
from binarytree import bst


# Create a random BST
# of any height
root = bst()
print('BST of any height : \n', root)

# Create a random BST of
# given height
root2 = bst(height = 2)
print('BST of given height : \n',
    root2)

# Create a random perfect
# BST of given height
root3 = bst(height = 2,
            is_perfect = True)
print('Perfect BST of given height : \n',
    root3)
```

Lab Tasks

Task 01 (tree.py):

Write a class that implements a simple binary **tree** data structure. That is a tree that consists of a group of nodes, each of which has an optional "left" and "right" child. The class needs to include the following:

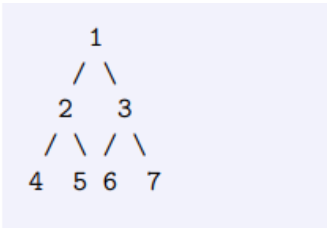
class Tree:

The class used to create objects representing trees. Each object will also represent one of the nodes of the tree. The constructor should take in one argument: the contents of that node. The constructor should have two optional arguments: left and right trees. If
given, these will be the left and right children of the new node. This allows for the joining of two trees into a single one with a new parent node.

self.left The left child of the node. May be None or a Tree object.

self.right The right child of the node. May be None or a Tree object.
self.datum The contents of the node.
def height(self) Returns the height of the tree that this node is the root of. This is equal to one more than the largest height of the children trees. If a child tree is None then its height is 0.
def add_item(self, item) Create a new Tree object and add it to the tree. The new object should be added to the child tree with the smallest height (remember that a None tree has a height of 0!). If there is a tie, favor the left tree.

Additionally, you should implement __iter__(self) and __next__(self) such that the tree can be used in a for loop. When used in a for loop, the nodes of the tree should be accessed in-order. This means that they should be accessed from left to right. That is, if we have a tree that looks like this:



Then the nodes should be accessed in the order 4, 2, 5, 1, 6, 3, 7. To accomplish this, you will need to keep track of where in the traversal the loop currently is, as well as raise StopIteration when necessary. Use treetest.py to test your code.

Task 02:

Implement the basic Functionality for **Binary Tree.**

1. Insertion
2. Deletion
3. Search
4. Preorder Traversal − Traverses a tree in a pre-order manner.
5. In order Traversal − Traverses a tree in an in-order manner.
6. Post order Traversal − Traverses a tree in a post-order manner.

Note:

Code also should check the Balance of tree.

Task 03.

Write a program that takes a string containing a postfix expression and builds a binary tree to represent the algebraic expression like that shown in Figure 1. visiting the nodes with a post-order traversal generates the expression AB+C*DE/− This is called **postfix notation**.

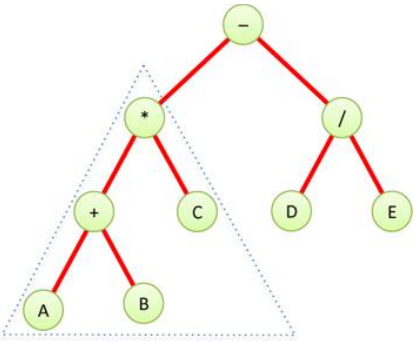


Figure 1. Binary tree representing an algebraic expression

Run your program on at least the following expressions:
a. 91 95 + 15 + 19 + 4 *
b. B B * A C 4 * * –
c. 42
d. A 57 # this should produce an exception
e. + / # this should produce an exception

Task 04. (BST Implementation)

In the introduction we used the binary search algorithm to find data stored in an array. This method is very effective, as each iteration reduced the number of items to search by one-half. However, since data was stored in an array, insertions and deletions were not efficient. Binary search trees store data in nodes that are linked in a tree-like fashion. For randomly inserted data, search time is O (lg n). Worst-case behavior occurs when ordered data is inserted. In this case the search time is O (n).

Design, Develop and Implement a menu driven Program in Python for the following operations with binary search trees:

1. Create a BST
2. Insertion of new element
3. Deletion of element
4. Search the BST for a given element (KEY) and report the appropriate message.
5. Sorting
6. Display
7. Exit

Note

Sorting should be done by traversing a BST but in an in-order manner.

Task 05:

Every binary tree can be represented as an array. The reverse of representing an array as a tree, however, works only for some arrays. The missing nodes of the tree are represented in the array cells as some predefined value—such as `None`—that cannot be a value stored at a tree node. If the root node is missing in the array, then the corresponding tree cannot be built.

Write a function that takes an array as input and tries to make a binary tree from its contents. Every cell that is not `None` is a value to store at a tree node. When you come across a node without a parent node (other than the root node), the function should raise an exception indicating that the tree cannot be built. Note that the result won't necessarily be a binary search tree, just a binary tree.

Hint: It's easier to work from the leaf nodes to the root, building nodes for each cell that is not `None` and storing the resulting node back in the same cell of the input array for retrieval when it is used as a subtree of a node on another level. Print the result of running the function on the following arrays where `n = None`. The values in the array can be stored as either the key or the value of the node because the tree won't be interpreted as a binary search tree.

```
[],
[n, n, n],
[55, 12, 71],
[55, 12, n, 4],
[55, 12, n, 4, n, n, n, n, 8, n, n, n, n, n, n, n, n, 6, n],
```

`[55, 12, n, n, n, n, 4, n, 8, n, n, n, n, n, n, n, n, 6, n]`

Happy Coding!

Deliverables

Comment your program heavily. Intelligent comments and a clean, readable formatting of your code account for 20% of your grade.

You should submit your codes and report as a compressed zip file. It should contain all files used in the exercises for this lab.

The submitted file should be names `cs250_firstname_lastname_lab12.zip`