Design Document Group No: 129
Problem Statement

Python uses Hash-Tables for implementing Dictionaries. You have to implement python-**dictionary** while using BST. But in this dictionary, the keys are numerical, and values could be either string or numerical. In this implementation **overload index-operator** ([]). For example, Dictionary [2] will give you the value stored at key=2. Also implement ".keys()" function and ".values()" like that of a dictionary. BST-Dictionary keys are numerical, but values could be a list, tuple, string, BST-Dictionary, or any other primitive data types. Make sure your searching is of order "log(n)". Commands given in input file should be able to run exactly (see example for details)

Remember following points:

1. Keys are always greater than 0.
2. You can research __getitem__ for operator-overloading.
3. Do not use in-built "dictionary" in python anywhere for this assignment.
4. DO-NOT use dictionary anywhere in the code.
5. DO-NOT copy from any website, other groups might copy as well and then it would be a plagiarism case.
6. No more test-cases would be provided, you'd have to consider all possible cases/scenarios for fail-free execution.

System overview

For dictionary implementation **BSTDictionary** and **Node** class is being created. The dictionary also implements operator overloading using **__setitem__** and **__getitem__**methods. Following table describe detailed information about **BSTDictionary** Class

Function Name	Description
`def __init__(self, allow_duplicates=True):`	The constructor method of BSTDictionary initializes the root node to None and allows duplicates by default.The allow_duplicates parameter can be set to False to disallow nodes with duplicate keys
`def get_height(self, node):`	Returns the heigh of given node
`def get_balance(self, node):`	Calculate the balance factor for a given node. Balance Factor=Height(Left Sub Tree)- Height(Right Sub Tree)
`def rotate_left(self, node):`	Performs Left Rotation
`def rotate_right(self, node):`	Performs Right Rotation
`def search(self, key):`	Perform Search for a given key and return associated Node data. If key is not found,"KeyError" is raised
`def containsKey(self,key):`	Returns True in case key is present otherwise false.
`def print_tree(self, fileName):`	# Print graphical representaton of Tree. This method uses the Graphviz library to create a visualization of the tree in the form of PDF File. Left Edges are represented in red color. Right Edges are reprented in Green Color. All generated files are created in Tree folder. If output file is already present, it will delete the old file and create the new file.
`def keys(self):`	Returns list of keys
`def values(self):`	Returns List of Values
`def __setitem__(self, key, value):`	Sets the value in Dictionary for a Given Key
`def __getitem__(self, key):`	Returns the value for a given key, otherwise raises KeyError

`def insert(self, key, data):`	Utility Method for inserting Key into Dictionary. Based on the value of allow_duplicates parameter allows insertion of duplicate key.
`def delete(self,key):`	Removes a given key from dictionary. If key is not found it will not delete anything otherwise delete the node.

Architecture Design

For implementing Dictionary, different data structures were evaluated BST and AVL Tree. After evaluating BST and AVL Tree performance for searching and inserting Elements AVL Tree is selected for Dictionary implementation:

1. Due to the balancing property, the insertion, deletion, and search operations take O (log n) in both the average and the worst cases. Therefore, AVL trees give us an edge over Binary Search Trees which have an O(n) time complexity in the worst-case scenario.
2. The space complexity of an AVL tree is O(n) in both the average and the worst case.

Implementation considerations

1. Keys can be only positive numbers greater than 0. `TEST_INVALID_KEY` is implemented to test all use cases related with keys.

Performance Test Data

As per below table we can clearly see, when the of n is increasing , insertion time increase however the search time is not increasing as much.

Total Elements	Insertion Time (milliseconds)	Avg Access Time (milliseconds)	$\log_2 x$
5000	28.569300018716604	0.0286192998 milliseconds	12.288
50000	413.10549998888746	0.4140854006	15.61
500000	6462.474800006021	6.4783421023	18.93
5000000	84458.78159999847	84.5177313018	22.253

When semi sorted data is being used to Create BST it is taking 13.88 milliseconds for 500 points however AVL Tree is taking 2.05 milliseconds. For 5000 semi sorted data, BST is throwing the error. “Maximum recursion depth exceeded” however AVL tree to continue to work without any issue.

Run Time Analysis

Function Name	Run time Analysis
`def __init__(self, allow_duplicates=True):`	O(1) Constant time
`def get_height(self, node):`	O(logn)
`def get_balance(self, node):`	O(logn)
`def rotate_left(self, node):`	O (1). Fixed number of rotations are performed. Which does not depend on height of left or right subtree
`def rotate_right(self, node):`	
`def search(self, key):`	O(logn)
`def __getitem__(self, key):`	
`def print_tree(self, fileName):`	O(n). A complete tree traversal is required

`def keys(self):`	
`def values(self):`	
`def __setitem__(self, key, value):`	O(logn)
`def insert(self, key, data):`	
`def containsKey(self,key):`	O(logn)
`def delete(self,key):`	O(logn)

Alternate Approach

The dictionary can be implemented using **Binary Search Tree** or using **Red Black Tree**.

- **Binary Search Tree:** the height of the tree is O(n). searching is not efficient where there are large of number of nodes present in BST. Also, BST is not a balanced tree. In BST insertion and deletion are easy, as there is no rotation is required. The code is present inside **AlternateSolution** directory.
- **Red Black TREE:** Red Black Tree is another data structure that can be used for implementing Dictionary. Insertion s and deletions are faster compared to AVL Tree, due to a smaller number of rotations. Searching is comparatively. The tree is less balanced compared to AVL Tree. It takes less processing for balancing; maximum two rotations are required. The code is present inside **AlternateSolution** directory.

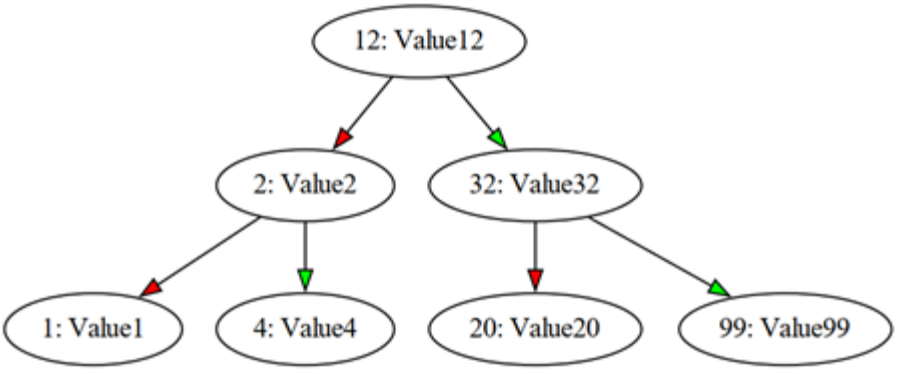
Further improvements

1. For storing keys, a separate List can be maintained, which will hold all the keys. This will reduce the time complexity of accessing all key values to O (1). However, this will require extra space. This can further be enhanced, using laze approach that is, on first call to Keys methods construct the List in O (n) complexity, and then while performing addition to dictionary, add keys to list. The same is applicable while performing deletion operation.
2. For storing values, a separate List can be maintained, which will hold all the values. This will reduce the time complexity of accessing all values to O (1). However, this will require extra space. This can further be enhanced, using laze approach that is, on first call to values methods construct the List in O (n) complexity, and then while performing addition to dictionary, add values to list. The same is applicable while performing deletion operation.

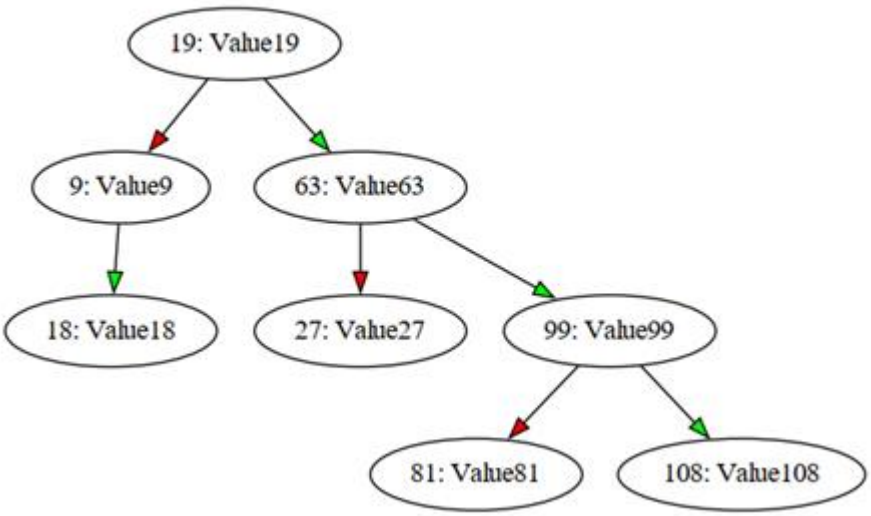
Dictionary Internal Data structure for various inputs

Red Arrow --> Left Child. **Green Arrow** --> Right Child

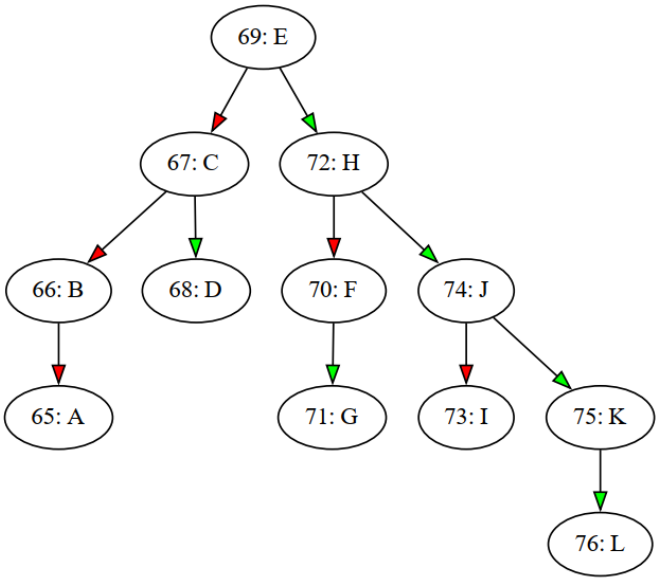
Input: **2, 1, 4, 20, 12, 99, 32**



Input: **63, 9, 19, 27, 18, 108, 99, 81**



Input: **'H', 'I', 'J', 'B', 'A', 'E', 'C', 'F', 'D','G', 'K', 'L'**. the input Is converted into ASCII Values and Dictionary is created.
ASCII Values [**72, 73, 74, 66, 65, 69, 67, 70, 68, 71, 75, 76**]



Input: **64, 1, 14, 26, 13, 110, 98, 85**