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 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. |

# 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.

# 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): |

# 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.
* 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.

# 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.