Dennis Qiu

Dr. Tompkins

CSC 231

13 November 2017

Lab8&9 Paper

PROCEDURE

1. The procedure is to study the Binary search tree and AVL search tree and search the big-O performance.
2. Write the Binary search tree and AVL search tree classes and implement the search methods. Group together data sets from dictionary.txt in two categories and different list sizes. Use 5000 size instead of 10000 due to the computer resource limitation and sizes 100, 1000, 10000, and 20000.
3. Then put in a random word list and ordered word list (ascending indexes).
4. Perform word searches on both tree types in different sizes and collect the search times. Save them in four separate CSV files:
   1. Searching a word in the random word list
   2. Searching a word that is not in the random word list
   3. Searching a word in the ordered word list
   4. Searching a word that is not in the ordered word list

Make sure to compare the recorded search times in each CSV file, along with O(Logn) and O(n) data.

1. Graph in Excel to show the big-O performances.

EXPECTED FINDINGS

1. The random list search should have the Big-O(n) performance, since the value of the nodes are random. The tree structure built with the keys will not help; it will need to go through all the nodes to the find the value.
2. The order list search should have better performance with AVL tree, since the value of the nodes has the same sorted order with the keys, value search can take advantage of the tree structure. For AVL trees, it should have Big-O(Logn) performance, as with the tree heights. For binary search tree, it may still be Big-O(n), since the tree height for the binary tree is same as the size of tree.
3. Searching for the word not in tree always has the worst time than searching for the word in the tree, since word not in tree search needs to traverse every node on the tree before it determines no node with this value was on the tree, so it’s the worst case of the searches. But since AVL tree has Logn performance, the difference may not be as obvious as the Binary search tree.

RESULTS

* See if result can be derived from each of the four cases using the data and graphs, which is the output collected from running the program:
  1. Searching a word that is in the random word list
  2. Searching a word that is not in the random word list
  3. Searching a word that is in the ordered word list
  4. Searching a word that is not in the ordered word list
* Expected Results Vs. Actual Results:
  1. Expected time for searching the word in tree in both random and ordered lists to be faster than the time for searching the word not in tree in both random and ordered lists
  2. Expected AVL search to be faster than BST search
  3. Actually, in word in random list for sizes 100 and 1000, BST search time is faster than AVL search time, which was unexpected, since AVL is supposed to be more efficient because of its balanced tree compared to BST’s unbalanced tree
  4. Actually, the search time for a word in tree is indeed faster than the search time for a word not in tree for both random and ordered lists
  5. Actually, in random and ordered lists for both kinds there’s no BST search for sizes 10000 and 20000, only AVL search instead
  6. Actually, in word not in random list the BST search time is indeed faster than AVL search time
  7. Actually, in ordered list for both kinds, BST search time is indeed faster than AVL search time
  8. Graphs:
  9. Output:

SEARCHING WORD IN TREE WITH RANDOM LIST

BinarySearchTree, height of tree: 100, size of tree: 100

yes, purflings is in tree, found at key 36

Searching for word "purflings" in size 100 tree took 0.000086 seconds

Based on time measured for current size 100: 0.000086 seconds, for next size 1000:

current : 0.000086

O(logN) : 0.000128

O(N) : 0.000855

O(NlogN) : 0.001283

O(N2) : 0.008553

AVLTree, height of tree: 7, size of tree: 100

yes, purflings is in tree, found at key 36

Searching for word "purflings" in size 100 tree took 0.000168 seconds

Based on time measured for current size 100: 0.000168 seconds, for next size 1000:

current : 0.000168

O(logN) : 0.000251

O(N) : 0.001676

O(NlogN) : 0.002514

O(N2) : 0.016763

BinarySearchTree, height of tree: 1000, size of tree: 1000

yes, purflings is in tree, found at key 36

Searching for word "purflings" in size 1000 tree took 0.000119 seconds

Based on time measured for current size 1000: 0.000119 seconds, for next size 10000:

current : 0.000119

O(logN) : 0.000159

O(N) : 0.001192

O(NlogN) : 0.001589

O(N2) : 0.011917

AVLTree, height of tree: 10, size of tree: 1000

yes, purflings is in tree, found at key 36

Searching for word "purflings" in size 1000 tree took 0.001460 seconds

Based on time measured for current size 1000: 0.001460 seconds, for next size 10000:

current : 0.001460

O(logN) : 0.001946

O(N) : 0.014597

O(NlogN) : 0.019462

O(N2) : 0.145966

Exception caught while building the BinarySearchTree with size 10000: maximum recursion depth exceeded

AVLTree, height of tree: 14, size of tree: 10000

yes, purflings is in tree, found at key 36

Searching for word "purflings" in size 10000 tree took 0.014516 seconds

Based on time measured for current size 10000: 0.014516 seconds, for next size 20000:

current : 0.014516

O(logN) : 0.015608

O(N) : 0.029031

O(NlogN) : 0.031216

O(N2) : 0.058063

Exception caught while building the BinarySearchTree with size 20000: maximum recursion depth exceeded

AVLTree, height of tree: 15, size of tree: 20000

yes, purflings is in tree, found at key 36

Searching for word "purflings" in size 20000 tree took 0.036127 seconds

SEARCHING WORD NOT IN TREE WITH RANDOM LIST

BinarySearchTree, height of tree: 100, size of tree: 100

no, dark is not found in tree

Searching for word "dark" in size 100 tree took 0.000175 seconds

Based on time measured for current size 100: 0.000175 seconds, for next size 1000:

current : 0.000175

O(logN) : 0.000263

O(N) : 0.001750

O(NlogN) : 0.002626

O(N2) : 0.017505

AVLTree, height of tree: 7, size of tree: 100

no, dark is not found in tree

Searching for word "dark" in size 100 tree took 0.000159 seconds

Based on time measured for current size 100: 0.000159 seconds, for next size 1000:

current : 0.000159

O(logN) : 0.000239

O(N) : 0.001591

O(NlogN) : 0.002386

O(N2) : 0.015908

BinarySearchTree, height of tree: 1000, size of tree: 1000

no, dark is not found in tree

Searching for word "dark" in size 1000 tree took 0.001747 seconds

Based on time measured for current size 1000: 0.001747 seconds, for next size 10000:

current : 0.001747

O(logN) : 0.002329

O(N) : 0.017470

O(NlogN) : 0.023294

O(N2) : 0.174703

AVLTree, height of tree: 10, size of tree: 1000

no, dark is not found in tree

Searching for word "dark" in size 1000 tree took 0.001467 seconds

Based on time measured for current size 1000: 0.001467 seconds, for next size 10000:

current : 0.001467

O(logN) : 0.001955

O(N) : 0.014665

O(NlogN) : 0.019553

O(N2) : 0.146650

Exception caught while building the BinarySearchTree with size 10000: maximum recursion depth exceeded

AVLTree, height of tree: 14, size of tree: 10000

no, dark is not found in tree

Searching for word "dark" in size 10000 tree took 0.015186 seconds

Based on time measured for current size 10000: 0.015186 seconds, for next size 20000:

current : 0.015186

O(logN) : 0.016329

O(N) : 0.030372

O(NlogN) : 0.032658

O(N2) : 0.060745

Exception caught while building the BinarySearchTree with size 20000: maximum recursion depth exceeded

AVLTree, height of tree: 15, size of tree: 20000

no, dark is not found in tree

Searching for word "dark" in size 20000 tree took 0.029050 seconds

SEARCHING WORD IN TREE WITH ORDERED LIST

BinarySearchTree, height of tree: 100, size of tree: 100

yes, abampere is in tree, found at key 36

Searching for word "abampere" in size 100 tree took 0.000197 seconds

Based on time measured for current size 100: 0.000197 seconds, for next size 1000:

current : 0.000197

O(logN) : 0.000296

O(N) : 0.001973

O(NlogN) : 0.002959

O(N2) : 0.019728

AVLTree, height of tree: 7, size of tree: 100

yes, abampere is in tree, found at key 36

Searching for word "abampere" in size 100 tree took 0.000083 seconds

Based on time measured for current size 100: 0.000083 seconds, for next size 1000:

current : 0.000083

O(logN) : 0.000125

O(N) : 0.000832

O(NlogN) : 0.001249

O(N2) : 0.008325

BinarySearchTree, height of tree: 1000, size of tree: 1000

yes, abampere is in tree, found at key 36

Searching for word "abampere" in size 1000 tree took 0.000121 seconds

Based on time measured for current size 1000: 0.000121 seconds, for next size 10000:

current : 0.000121

O(logN) : 0.000162

O(N) : 0.001214

O(NlogN) : 0.001619

O(N2) : 0.012145

AVLTree, height of tree: 10, size of tree: 1000

yes, abampere is in tree, found at key 36

Searching for word "abampere" in size 1000 tree took 0.000071 seconds

Based on time measured for current size 1000: 0.000071 seconds, for next size 10000:

current : 0.000071

O(logN) : 0.000095

O(N) : 0.000713

O(NlogN) : 0.000950

O(N2) : 0.007127

Exception caught while building the BinarySearchTree with size 10000: maximum recursion depth exceeded

AVLTree, height of tree: 14, size of tree: 10000

yes, abampere is in tree, found at key 36

Searching for word "abampere" in size 10000 tree took 0.000083 seconds

Based on time measured for current size 10000: 0.000083 seconds, for next size 20000:

current : 0.000083

O(logN) : 0.000090

O(N) : 0.000166

O(NlogN) : 0.000179

O(N2) : 0.000333

Exception caught while building the BinarySearchTree with size 20000: maximum recursion depth exceeded

AVLTree, height of tree: 15, size of tree: 20000

yes, abampere is in tree, found at key 36

Searching for word "abampere" in size 20000 tree took 0.000105 seconds

SEARCHING WORD NOT IN TREE WITH ORDERED LIST

BinarySearchTree, height of tree: 100, size of tree: 100

no, dark is not found in tree

Searching for word "dark" in size 100 tree took 0.000193 seconds

Based on time measured for current size 100: 0.000193 seconds, for next size 1000:

current : 0.000193

O(logN) : 0.000290

O(N) : 0.001933

O(NlogN) : 0.002899

O(N2) : 0.019329

AVLTree, height of tree: 7, size of tree: 100

no, dark is not found in tree

Searching for word "dark" in size 100 tree took 0.000026 seconds

Based on time measured for current size 100: 0.000026 seconds, for next size 1000:

current : 0.000026

O(logN) : 0.000039

O(N) : 0.000262

O(NlogN) : 0.000393

O(N2) : 0.002623

BinarySearchTree, height of tree: 1000, size of tree: 1000

no, dark is not found in tree

Searching for word "dark" in size 1000 tree took 0.001607 seconds

Based on time measured for current size 1000: 0.001607 seconds, for next size 10000:

current : 0.001607

O(logN) : 0.002142

O(N) : 0.016068

O(NlogN) : 0.021424

O(N2) : 0.160677

AVLTree, height of tree: 10, size of tree: 1000

no, dark is not found in tree

Searching for word "dark" in size 1000 tree took 0.000073 seconds

Based on time measured for current size 1000: 0.000073 seconds, for next size 10000:

current : 0.000073

O(logN) : 0.000097

O(N) : 0.000730

O(NlogN) : 0.000973

O(N2) : 0.007298

Exception caught while building the BinarySearchTree with size 10000: maximum recursion depth exceeded

AVLTree, height of tree: 14, size of tree: 10000

no, dark is not found in tree

Searching for word "dark" in size 10000 tree took 0.000090 seconds

Based on time measured for current size 10000: 0.000090 seconds, for next size 20000:

current : 0.000090

O(logN) : 0.000096

O(N) : 0.000179

O(NlogN) : 0.000193

O(N2) : 0.000358

Exception caught while building the BinarySearchTree with size 20000: maximum recursion depth exceeded

AVLTree, height of tree: 15, size of tree: 20000

no, dark is not found in tree

Searching for word "dark" in size 20000 tree took 0.000100 seconds

Process finished with exit code 0

APPENDICES

treeNode.py

\_\_author\_\_ = **'Dennis Qiu'  
  
class** TreeNode():  
 *"""  
 Description: Implement a tree node with the following interface  
 functions:  
 \_\_init\_\_(k, v, lc, rc, p)  
 hasLeftChild()  
 hasRightChild()  
 isLeftChild()  
 isRightChild()  
 isRoot()  
 isLeaf()  
 hasAnyChildren()  
 hasBothChildren()  
 replaceNodeData(k, v, lc, rc)  
 """* **def** \_\_init\_\_(self,key,val,left=**None**,right=**None**,  
 parent=**None**):  
 *"""  
 Initializes TreeNode.* **:param** *key: indices of list in TreeNode* **:param** *val: additional info contained in a node* **:param** *left: node to the left below its parent node (incoming edges)-->node on left subtree* **:param** *right: node to the right below its parent node (incoming edges)--> node on right subtree* **:param** *parent: "Parent" of all nodes it connects with outgoing edges* **:return***: reference for self  
 """* self.key = key  
 self.payload = val  
 self.leftChild = left  
 self.rightChild = right  
 self.parent = parent  
 self.balanceFactor = 0  
  
 **def** hasLeftChild(self):  
 *"""  
 The subtree containing the left child node* **:return***: node for left child  
 """* **return** self.leftChild  
  
 **def** hasRightChild(self):  
 *"""  
 The subtree containing the right child node* **:return***: node for right child  
 """* **return** self.rightChild  
  
 **def** isLeftChild(self):  
 *"""  
 If node is the lc node* **:return***: parent node and lc node attached to parent node  
 """* **return** self.parent **and** self.parent.leftChild == self  
  
 **def** isRightChild(self):  
 *"""  
 If node is the rc node* **:return***: parent node and rc node attached to parent node  
 """* **return** self.parent **and** self.parent.rightChild == self  
  
 **def** isRoot(self):  
 *"""  
 (The root of the tree) node in tree with no incoming edges* **:return***: does not return parent node  
 """* **return not** self.parent  
  
 **def** isLeaf(self):  
 *"""  
 Node that has no children* **:return***: does not return node for rc or node for lc  
 """* **return not** (self.rightChild **or** self.leftChild)  
  
 **def** hasAnyChildren(self):  
 *"""  
 If a node has a lc node or rc node* **:return***: either node for rc or node for lc  
 """* **return** self.rightChild **or** self.leftChild  
  
 **def** hasBothChildren(self):  
 *"""  
 If a node has both lc and rc nodes* **:return***: both nodes for lc and rc  
 """* **return** self.rightChild **and** self.leftChild  
  
 **def** replaceNodeData(self,key,value,lc,rc):  
 *"""  
 Replaces original node data with new node data* **:param** *key: indices of list* **:param** *value: additional info contained in a node* **:param** *lc: node to the left below its parent node (incoming edges)-->node on left subtree* **:param** *rc: node to the right below its parent node (incoming edges)--> node on right subtree* **:return***: None  
 """* self.key = key  
 self.payload = value  
 self.leftChild = lc  
 self.rightChild = rc  
 **if** self.hasLeftChild():  
 self.leftChild.parent = self  
 **if** self.hasRightChild():  
 self.rightChild.parent = self  
  
 **def** height(self, node):  
 *"""  
 Height of any given node in tree* **:param** *node: node in tree* **:return***: height of any given node  
 """* **return** 1 + max(self.height(node.leftChild) **if** node.leftChild **is not None else** 0,  
 self.height(node.rightChild) **if** node.rightChild **is not None else** 0)

Help on class TreeNode in module \_\_main\_\_:

class TreeNode(builtins.object)

| Description: Implement a tree node with the following interface

| functions:

| \_\_init\_\_(k, v, lc, rc, p)

| hasLeftChild()

| hasRightChild()

| isLeftChild()

| isRightChild()

| isRoot()

| isLeaf()

| hasAnyChildren()

| hasBothChildren()

| replaceNodeData(k, v, lc, rc)

|

| Methods defined here:

|

| \_\_init\_\_(self, key, val, left=None, right=None, parent=None)

| Initializes TreeNode.

| :param key: indices of list in TreeNode

| :param val: additional info contained in a node

| :param left: node to the left below its parent node (incoming edges)-->node on left subtree

| :param right: node to the right below its parent node (incoming edges)--> node on right subtree

| :param parent: "Parent" of all nodes it connects with outgoing edges

| :return: reference for self

|

| hasAnyChildren(self)

| If a node has a lc node or rc node

| :return: either node for rc or node for lc

|

| hasBothChildren(self)

| If a node has both lc and rc nodes

| :return: both nodes for lc and rc

|

| hasLeftChild(self)

| The subtree containing the left child node

| :return: node for left child

|

| hasRightChild(self)

| The subtree containing the right child node

| :return: node for right child

|

| height(self, node)

| Height of any given node in tree

| :param node: node in tree

| :return: height of any given node

|

| isLeaf(self)

| Node that has no children

| :return: does not return node for rc or node for lc

|

| isLeftChild(self)

| If node is the lc node

| :return: parent node and lc node attached to parent node

|

| isRightChild(self)

| If node is the rc node

| :return: parent node and rc node attached to parent node

|

| isRoot(self)

| (The root of the tree) node in tree with no incoming edges

| :return: does not return parent node

|

| replaceNodeData(self, key, value, lc, rc)

| Replaces original node data with new node data

| :param key: indices of list

| :param value: additional info contained in a node

| :param lc: node to the left below its parent node (incoming edges)-->node on left subtree

| :param rc: node to the right below its parent node (incoming edges)--> node on right subtree

| :return: None

|

| ----------------------------------------------------------------------

| Data descriptors defined here:

|

| \_\_dict\_\_

| dictionary for instance variables (if defined)

|

| \_\_weakref\_\_

| list of weak references to the object (if defined)

>>>

binarySearchTree.py

\_\_author\_\_ = **'Dennis Qiu'  
from** treeNode **import** TreeNode  
  
**class** BinarySearchTree(TreeNode):  
 *"""  
 Description: Implement a binary search tree with the following interface  
 functions:  
 \_\_contains\_\_(y) <==> y in x  
 \_\_getitem\_\_(y) <==> x[y]  
 \_\_init\_\_()  
 \_\_len\_\_() <==> len(x)  
 \_\_setitem\_\_(k,v) <==> x[k] = v, raises KeyError Exception  
 clear()  
 get(k)  
 height()  
 items()  
 keys()  
 values()  
 put(k,v)  
 in  
 del <==>  
 """* **def** \_\_init\_\_(self):  
 *"""  
 Initializes BinarySearchTree.* **:return***: reference for self  
 """* self.root = **None** self.size = 0  
  
 **def** \_\_contains\_\_(self, key):  
 *"""  
 Simply calls get.  
 Impliments in operation, and overloads in operator* **:param** *key: index of list in BinarySearchTree* **:return***: Return True if get returns a value, or False if it returns None  
 """* **if** self.\_get(key, self.root):  
 **return True  
 else**:  
 **return False  
  
 def** \_\_getitem\_\_(self, key):  
 *"""  
 Same logic as \_\_setitem\_\_.  
 Calls get() method.* **:param** *key: indice of list in BinarySearchTree* **:return***: key(a node in tree)  
 """* **return** self.get(key)  
  
 **def** \_\_len\_\_(self):  
 *"""  
 Length of list* **:return***: Instance that returns length of list  
 """* **return** self.size  
  
 **def** \_\_iter\_\_(self):  
 *"""* **:return***:  
 """* **return** self.root.\_\_iter\_\_()  
  
 **def** \_\_setitem\_\_(self, k, v):  
 *"""  
 Calls put() method.  
 Overloads [] operator to allow access for writing statements* **:param** *k: indices of list in BinarySearchTree* **:param** *v: additional info contained in a node* **:return***: None  
 """* self.put(k, v)  
  
 **def** clear(self):  
 *"""  
 Removes all items from list.  
 Empties (clears) tree by re-setting root to None and size to 0* **:return***: None  
 """* self.root = **None** self.size = 0  
  
 **def** get(self, key):  
 *"""  
 Same logic as put() method.* **:param** *key: index of list in BinarySearchTree* **:return***: Returns payload, otherwise none/nothing else  
 """* **if** self.root:  
 res = self.\_get(key, self.root)  
 **if** res:  
 **return** res.payload  
 **else**:  
 **return None  
 else**:  
 **return None  
  
 def** \_get(self, key, currentNode):  
 *"""  
 Same logic as \_put() method.* **:param** *key: index of list in BinarySearchTree* **:param** *currentNode: node that is currently compared to in tree* **:return***:  
 """* **if not** currentNode:  
 **return None  
 elif** currentNode.key == key:  
 **return** currentNode  
 **elif** key < currentNode.key:  
 **return** self.\_get(key, currentNode.leftChild)  
 **else**:  
 **return** self.\_get(key, currentNode.rightChild)  
  
 **def** height(self, node):  
 *"""  
 Height of tree root* **:param** *node: root node* **:return***: height of tree  
 """* **if not** self.root:  
 **return** 0  
 **else**:  
 **return** node.height(node)  
  
 **def** items(self):  
 *"""* **:return***:  
 """* Nodes = []  
 **if not** self.root:  
 **return None  
 else**:  
 self.\_items(Nodes, self.root)  
 **while None in** Nodes:  
 Nodes.remove(**None**)  
 **return** Nodes  
  
 **def** \_items(self, items, node):  
 *"""* **:param** *items: items in list* **:param** *node: node in tree* **:return***: None  
 """* **if** node == **None**:  
 **return  
 else**:  
 items.append(node)  
 **if** node.hasLeftChild:  
 items.append(self.\_items(items, node.leftChild))  
 **if** node.hasRightChild:  
 items.append(self.\_items(items, node.rightChild))  
  
 **def** keys(self):  
 *"""  
 Keys in list, or nodes in tree* **:return***: None  
 """* Keys = []  
 **if** self.root != **None**:  
 self.\_keys(Keys, self.root)  
 **while None in** Keys:  
 Keys.remove(**None**)  
 **return** Keys  
 **else**:  
 **return None  
  
 def** \_keys(self, allkeys, node):  
 *"""* **:param** *allkeys: all keys in list* **:param** *node: node in tree* **:return***: None  
 """* **if** node == **None**:  
 **return  
 else**:  
 allkeys.append(node.key)  
 **if** node.hasLeftChild:  
 allkeys.append(self.\_keys(allkeys, node.leftChild))  
 **if** node.hasRightChild:  
 allkeys.append(self.\_keys(allkeys, node.rightChild))  
  
 **def** values(self):  
 *"""* **:return***:  
 """* Values = []  
 **for** k **in** self.keys():  
 Values.append(self.get(k))  
 **return** Values  
  
  
 **def** put(self, key, val):  
 *"""  
 Builds Binary Search Tree.  
 Checks to see if the tree has a root.  
 If no root, creates a new TreeNode and installs it as root of tree.* **:param** *key: index of list in BinarySearchTree* **:param** *val: additional info contained in a node* **:return***: None  
 """* **if** self.root:  
 self.\_put(key, val, self.root)  
 **else**:  
 self.root = TreeNode(key, val)  
 self.size = self.size + 1  
  
 **def** \_put(self, key, val, currentNode):  
 *"""  
 If there's already a root node in place  
 Compare new key with current node in tree, both subtrees,  
 When there's no left or right child to search  
 Install new node in this position* **:param** *key: index of list in BinarySearchTree* **:param** *val: additional info contained in a node* **:param** *currentNode: node that is currently compared to in tree* **:return***: None  
 """* **if** key < currentNode.key:  
 **if** currentNode.hasLeftChild():  
 self.\_put(key, val, currentNode.leftChild)  
 **else**:  
 currentNode.leftChild = TreeNode(key, val, parent=currentNode)  
 **elif** key > currentNode.key:  
 **if** currentNode.hasRightChild():  
 self.\_put(key, val, currentNode.rightChild)  
 **else**:  
 currentNode.rightChild = TreeNode(key, val, parent=currentNode)  
 **else**:  
 currentNode.replaceNodeData(key, val, currentNode.leftChild, currentNode.rightChild)  
 self.size -= 1  
  
 **def** delete(self, key):  
 *"""  
 Use \_get method to find node TreeNode to be removed.  
 del operator raises KeyError if key is not found.* **:param** *key: index of list in BinarySearchTree to be removed* **:return***: None  
 """* **if** self.size > 1:  
 nodeToRemove = self.\_get(key, self.root)  
 **if** nodeToRemove:  
 self.remove(nodeToRemove)  
 self.size = self.size - 1  
 **else**:  
 **raise** KeyError(**'Error, key not in tree'**)  
 **elif** self.size == 1 **and** self.root.key == key:  
 self.root = **None** self.size = self.size - 1  
 **else**:  
 **raise** KeyError(**'Error, key not in tree'**)  
  
 **def** \_\_delitem\_\_(self, key):  
 *"""  
 Calls delete() method. Deletes the key.* **:param** *key: index of list in BinarySearchTree to be deleted* **:return***: None  
 """* self.delete(key)  
  
 **def** set\_balanceFactor(self):  
 *"""* **:return***:  
 """* **for** n **in** self.items():  
 **if** n.leftChild == **None**:  
 lh = 0  
 **else**:  
 lh = n.leftChild.height(n.leftChild)  
 **if** n.rightChild == **None**:  
 rh = 0  
 **else**:  
 rh = n.rightChild.height(n.rightChild)  
 n.balanceFactor = lh - rh  
  
 **def** search\_random\_list(self, val):  
 *"""* **:param** *val:* **:return***:  
 """* **if not** self.root:  
 **return None  
 else**:  
 self.Key\_found = **None** self.\_searchRandom(val, self.root)  
 **return** self.Key\_found  
  
 **def** \_searchRandom(self, val, node):  
 *"""* **:param** *val:* **:param** *node:* **:return***: None  
 """* **if** node == **None**:  
 **return  
 else**:  
 **if** node.payload == val:  
 self.Key\_found = node.key  
 **return  
 if** node.hasLeftChild:  
 self.\_searchRandom(val, node.leftChild)  
 **if** node.hasRightChild:  
 self.\_searchRandom(val, node.rightChild)  
  
 **def** search\_ordered\_list(self, val):  
 *"""* **:param** *val:* **:return***:  
 """* **if** self.root:  
 self.Key\_found = **None** self.\_searchOrdered(val, self.root)  
 **return** self.Key\_found  
 **else**:  
 **return None  
  
 def** \_searchOrdered(self, val, currentNode):  
 *"""* **:param** *val:* **:param** *currentNode:* **:return***:  
 """* **if** val < currentNode.payload:  
 **if** currentNode.hasLeftChild():  
 self.\_searchOrdered(val, currentNode.leftChild)  
 **else**:  
 **return  
 elif** val > currentNode.payload:  
 **if** currentNode.hasRightChild():  
 self.\_searchOrdered(val, currentNode.rightChild)  
 **else**:  
 **return  
 else**:  
 self.Key\_found = currentNode.key  
 **return  
def** main():  
 t = BinarySearchTree()  
 print(**'Empty tree, height {}, size {}'**.format(t.height(t.root), len(t)))  
 t[7] = **'seven'** print(**'t[7] = "seven", height {}, size {}'**.format(t.height(t.root), len(t)))  
 t[2] = **'two'** print(**'t[2] = "two", height {}, size {}'**.format(t.height(t.root), len(t)))  
 t[9] = **'nine'** print(**'t[9] = "nine", height {}, size {}'**.format(t.height(t.root), len(t)))  
 t[10] = **'ten'** print(**'t[10] = "ten", height {}, size {}'**.format(t.height(t.root), len(t)))  
 t[1] = **'one'** print(**'t[1] = "one", height {}, size {}'**.format(t.height(t.root), len(t)))  
 t[3] = **'three'** print(**'t[3] = "three", height {}, size {}'**.format(t.height(t.root), len(t)))  
 t.clear()  
 t[7] = **'2ndseven'** print(**'t[7] = "2ndseven", cleared tree, height {}, size {}'**.format(t.height(t.root), len(t)))  
 print(**'\t'**)  
  
 print(**'keys in t: {}'**.format(t.keys()))  
 print(**'values in t: {}'**.format(t.values()))  
 nodes = t.items()  
 t.set\_balanceFactor()  
 **for** i **in** nodes:  
 print(**'key = "{}", height {}, balanceFactor {}'**.format(i.key, i.height(i), i.balanceFactor))  
 print(**'\t'**)  
  
 print(**'{} items in t: {}'**.format(len(nodes), nodes))  
 t[7] = **'2ndseven'** print(**'t[7] = "{}", height {}, size {}'**.format(t[7], t.height(t.root), len(t)))  
  
 t2 = BinarySearchTree()  
 names = [**'zero'**, **'one'**, **'two'**, **'three'**, **'four'**, **'five'**, **'six'**, **'seven'**, **'eight'**, **'nine'**]  
 **for** k, v **in** enumerate(names):  
 t2[k] = v  
 print(**'t2[{}] = "{}", height {}, size {}'**.format(k, t2[k], t2.height(t2.root), len(t2)))  
 print(**'keys in t2: {}'**.format(t2.keys()))  
 print(**'\t'**)  
  
 **for** v **in** (5, 25):  
 **if** v **in** t2:  
 print(**'yes, {} is in t2, t2[{}] = "{}"'**.format(v, v, t2[v]))  
 **else**:  
 print(**"no, {} is not in t2"**.format(v))  
 print(**'\t'**)  
  
 t2.set\_balanceFactor()  
 **for** i **in** t2.items():  
 print(**'key = "{}", height {}, balanceFactor {}'**.format(i.key, i.height(i), i.balanceFactor))  
  
**if** \_\_name\_\_ == **'\_\_main\_\_'**:  
 main()

Help on class BinarySearchTree in module \_\_main\_\_:

class BinarySearchTree(treeNode.TreeNode)

| Description: Implement a binary search tree with the following interface

| functions:

| \_\_contains\_\_(y) <==> y in x

| \_\_getitem\_\_(y) <==> x[y]

| \_\_init\_\_()

| \_\_len\_\_() <==> len(x)

| \_\_setitem\_\_(k,v) <==> x[k] = v, raises KeyError Exception

| clear()

| get(k)

| height()

| items()

| keys()

| values()

| put(k,v)

| in

| del <==>

|

| Method resolution order:

| BinarySearchTree

| treeNode.TreeNode

| builtins.object

|

| Methods defined here:

|

| \_\_contains\_\_(self, key)

| Simply calls get.

| Impliments in operation, and overloads in operator

| :param key: index of list in BinarySearchTree

| :return: Return True if get returns a value, or False if it returns None

|

| \_\_delitem\_\_(self, key)

| Calls delete() method. Deletes the key.

| :param key: index of list in BinarySearchTree to be deleted

| :return: None

|

| \_\_getitem\_\_(self, key)

| Same logic as \_\_setitem\_\_.

| Calls get() method.

| :param key: indice of list in BinarySearchTree

| :return: key(a node in tree)

|

| \_\_init\_\_(self)

| Initializes BinarySearchTree.

| :return: reference for self

|

| \_\_iter\_\_(self)

| :return:

|

| \_\_len\_\_(self)

| Length of list

| :return: Instance that returns length of list

|

| \_\_setitem\_\_(self, k, v)

| v raises KeyError Exception.

| Calls put() method.

| Overloads [] operator to allow access for writing statements

| :param k: indices of list in BinarySearchTree

| :param v: additional info contained in a node

| :return: None

|

| clear(self)

| Removes all items from list.

| Empties (clears) tree by re-setting root to None and size to 0

| :return: None

|

| delete(self, key)

| Use \_get method to find node TreeNode to be removed.

| del operator raises KeyError if key is not found.

| :param key: index of list in BinarySearchTree to be removed

| :return: None

|

| get(self, key)

| Same logic as put() method.

| :param key: index of list in BinarySearchTree

| :return: Returns payload, otherwise none/nothing else

|

| height(self, node)

| Height of tree root

| :param node: root node

| :return: height of tree

|

| items(self)

| :return:

|

| keys(self)

| Keys in list, or nodes in tree

| :return: None

|

| put(self, key, val)

| Builds Binary Search Tree.

| Checks to see if the tree has a root.

| If no root, creates a new TreeNode and installs it as root of tree.

| :param key: index of list in BinarySearchTree

| :param val: additional info contained in a node

| :return: None

|

| search\_ordered\_list(self, val)

| :param val:

| :return:

|

| search\_random\_list(self, val)

| :param val:

| :return:

|

| set\_balanceFactor(self)

| :return:

|

| values(self)

| :return:

|

| ----------------------------------------------------------------------

| Methods inherited from treeNode.TreeNode:

|

| hasAnyChildren(self)

| If a node has a lc node or rc node

| :return: either node for rc or node for lc

|

| hasBothChildren(self)

| If a node has both lc and rc nodes

| :return: both nodes for lc and rc

|

| hasLeftChild(self)

| The subtree containing the left child node

| :return: node for left child

|

| hasRightChild(self)

| The subtree containing the right child node

| :return: node for right child

|

| isLeaf(self)

| Node that has no children

| :return: does not return node for rc or node for lc

|

| isLeftChild(self)

| If node is the lc node

| :return: parent node and lc node attached to parent node

|

| isRightChild(self)

| If node is the rc node

| :return: parent node and rc node attached to parent node

|

| isRoot(self)

| (The root of the tree) node in tree with no incoming edges

| :return: does not return parent node

|

| replaceNodeData(self, key, value, lc, rc)

| Replaces original node data with new node data

| :param key: indices of list

| :param value: additional info contained in a node

| :param lc: node to the left below its parent node (incoming edges)-->node on left subtree

| :param rc: node to the right below its parent node (incoming edges)--> node on right subtree

| :return: None

|

| ----------------------------------------------------------------------

| Data descriptors inherited from treeNode.TreeNode:

|

| \_\_dict\_\_

| dictionary for instance variables (if defined)

|

| \_\_weakref\_\_

| list of weak references to the object (if defined)

>>>

AVLtree.py

\_\_author\_\_ = **'Dennis Qiu'  
from** treeNode **import** TreeNode  
**from** binarySearchTree **import** BinarySearchTree  
  
**class** AVLTree(BinarySearchTree):  
 *"""  
 Description: Implement a AVL tree with the following interface  
 functions:  
 \_\_init\_\_()  
 \_put(k, v, cN)  
 updateBalance(n)  
 rotateLeft(r)  
 rebalance(n)  
 """* **def** \_\_init\_\_(self):  
 *"""  
 Instantiates initialize from BinarySearchTree  
 """* super().\_\_init\_\_()  
  
 **def** \_put(self, key, val, currentNode):  
 *"""  
 Builds AVL Tree.  
 Checks to see if the tree has a root.  
 If no root, creates a new TreeNode and installs it as root of tree.  
 Updates balance factor of parent* **:param** *key: index of list in AVLTree* **:param** *val: additional info contained in a node* **:param** *currentNode: node currently compared with key* **:return***: None  
 """* **if** key < currentNode.key:  
 **if** currentNode.hasLeftChild():  
 self.\_put(key, val, currentNode.leftChild)  
 **else**:  
 currentNode.leftChild = TreeNode(key, val, parent=currentNode)  
 self.updateBalance(currentNode.leftChild)  
 **else**:  
 **if** currentNode.hasRightChild():  
 self.\_put(key, val, currentNode.rightChild)  
 **else**:  
 currentNode.rightChild = TreeNode(key, val, parent=currentNode)  
 self.updateBalance(currentNode.rightChild)  
  
 **def** updateBalance(self, node):  
 *"""  
 Updating to parents.  
 If current node is out of balance enough, rebalance.  
 If current node is not out of balance, adjust balance factor of parent.  
 If bf is zero, continue up the tree towards root.* **:param** *node: node being balanced* **:return***: None  
 """* **if** node.balanceFactor > 1 **or** node.balanceFactor < -1:  
 self.rebalance(node)  
 **return  
 if** node.parent != **None**:  
 **if** node.isLeftChild():  
 node.parent.balanceFactor += 1  
 **elif** node.isRightChild():  
 node.parent.balanceFactor -= 1  
 **if** node.parent.balanceFactor != 0:  
 self.updateBalance(node.parent)  
  
 **def** rotateLeft(self, rotRoot):  
 *"""  
 Promotes left rotation.* **:param** *rotRoot: Right to left child* **:return***: None  
 """* newRoot = rotRoot.rightChild  
 rotRoot.rightChild = newRoot.leftChild  
 **if** newRoot.leftChild != **None**:  
 newRoot.leftChild.parent = rotRoot  
 newRoot.parent = rotRoot.parent  
 **if** rotRoot.isRoot():  
 self.root = newRoot  
 **else**:  
 **if** rotRoot.isLeftChild():  
 rotRoot.parent.leftChild = newRoot  
 **else**:  
 rotRoot.parent.rightChild = newRoot  
 newRoot.leftChild = rotRoot  
 rotRoot.parent = newRoot  
 rotRoot.balanceFactor = rotRoot.balanceFactor + 1 - min(newRoot.balanceFactor, 0)  
 newRoot.balanceFactor = newRoot.balanceFactor + 1 + max(rotRoot.balanceFactor, 0)  
  
 **def** rotateRight(self, rotRoot):  
 *"""  
 Promotes right rotation.* **:param** *rotRoot: Left to right child* **:return***: None  
 """* newRoot = rotRoot.leftChild  
 rotRoot.leftChild = newRoot.rightChild  
 **if** newRoot.rightChild != **None**:  
 newRoot.rightChild.parent = rotRoot  
 newRoot.parent = rotRoot.parent  
 **if** rotRoot.isRoot():  
 self.root = newRoot  
 **else**:  
 **if** rotRoot.rightChild():  
 rotRoot.parent.rightChild = newRoot  
 **else**:  
 rotRoot.parent.leftChild = newRoot  
 newRoot.rightChild = rotRoot  
 rotRoot.parent = newRoot  
 rotRoot.balanceFactor = rotRoot.balanceFactor + 1 - min(newRoot.balanceFactor, 0)  
 newRoot.balanceFactor = newRoot.balanceFactor + 1 + max(rotRoot.balanceFactor, 0)  
  
 **def** rebalance(self, node):  
 *"""  
 Rebalance node if it is out of balance* **:param** *node: node being balanced* **:return***: None  
 """* **if** node.balanceFactor < 0:  
 **if** node.rightChild.balanceFactor > 0:  
 self.rotateRight(node.rightChild)  
 self.rotateLeft(node)  
 **else**:  
 self.rotateLeft(node)  
 **elif** node.balanceFactor > 0:  
 **if** node.leftChild.balanceFactor < 0:  
 self.rotateLeft(node.leftChild)  
 self.rotateRight(node)  
 **else**:  
 self.rotateRight(node)  
  
**def** main():  
 t = AVLTree()  
 print(**'Empty tree, height {}, size {}'**.format(t.height(t.root), len(t)))  
 t[7] = **'seven'** print(**'t[7] = "seven", height {}, size {}'**.format(t.height(t.root), len(t)))  
 t[2] = **'two'** print(**'t[2] = "two", height {}, size {}'**.format(t.height(t.root), len(t)))  
 t[9] = **'nine'** print(**'t[9] = "nine", height {}, size {}'**.format(t.height(t.root), len(t)))  
 t[10] = **'ten'** print(**'t[10] = "ten", height {}, size {}'**.format(t.height(t.root), len(t)))  
 t[1] = **'one'** print(**'t[1] = "one", height {}, size {}'**.format(t.height(t.root), len(t)))  
 t[3] = **'three'** print(**'t[3] = "three", height {}, size {}'**.format(t.height(t.root), len(t)))  
 print(**'\t'**)  
  
 print(**'keys in t: {}'**.format(t.keys()))  
 print(**'values in t: {}'**.format(t.values()))  
 nodes = t.items()  
 print(**'{} items in t: {}'**.format(len(nodes), nodes))  
 print(**'\t'**)  
  
 t[7] = **'2ndseven'** print(**'t[7] = "{}", height {}, size {}'**.format(t[7], t.height(t.root), len(t)))  
  
 t2 = AVLTree()  
 names = [**'zero'**, **'one'**, **'two'**, **'three'**, **'four'**, **'five'**, **'six'**, **'seven'**, **'eight'**, **'nine'**]  
 **for** k, v **in** enumerate(names):  
 t2[k] = v  
 print(**'t2[{}] = "{}", height {}, size {}'**.format(k, t2[k], t2.height(t2.root), len(t2)))  
  
 print(**'\t'**)  
 print(**'keys in t2: {}'**.format(t2.keys()))  
  
 **for** v **in** (5, 25):  
 **if** v **in** t2:  
 print(**'yes, {} is in t2, t2[{}] = "{}"'**.format(v, v, t2[v]))  
 **else**:  
 print(**"no, {} is not in t2"**.format(v))  
  
**if** \_\_name\_\_ == **'\_\_main\_\_'**:  
 main()

Help on class AVLTree in module \_\_main\_\_:

class AVLTree(binarySearchTree.BinarySearchTree)

| Method resolution order:

| AVLTree

| binarySearchTree.BinarySearchTree

| treeNode.TreeNode

| builtins.object

|

| Methods defined here:

|

| \_\_init\_\_(self)

|

| rebalance(self, node)

| :param node:

| :return:

|

| rotateLeft(self, rotRoot)

| :param rotRoot:

| :return:

|

| updateBalance(self, node)

| :param node:

| :return:

|

| ----------------------------------------------------------------------

| Methods inherited from binarySearchTree.BinarySearchTree:

|

| \_\_contains\_\_(self, key)

| Simply calls get.

| Impliments in operation, and overloads in operator

| :param key: index of list in BinarySearchTree

| :return: Return True if get returns a value, or False if it returns None

|

| \_\_delitem\_\_(self, key)

| Calls delete() method. Deletes the key.

| :param key: index of list in BinarySearchTree to be deleted

| :return: None

|

| \_\_getitem\_\_(self, key)

| Same logic as \_\_setitem\_\_.

| Calls get() method.

| :param key: indice of list in BinarySearchTree

| :return: key(a node in tree)

|

| \_\_iter\_\_(self)

| :return:

|

| \_\_len\_\_(self)

| Length of list

| :return: Instance that returns length of list

|

| \_\_setitem\_\_(self, k, v)

| v raises KeyError Exception.

| Calls put() method.

| Overloads [] operator to allow access for writing statements

| :param k: indices of list in BinarySearchTree

| :param v: additional info contained in a node

| :return: None

|

| clear(self)

| Removes all items from list.

| Empties (clears) tree by re-setting root to None and size to 0

| :return: None

|

| delete(self, key)

| Use \_get method to find node TreeNode to be removed.

| del operator raises KeyError if key is not found.

| :param key: index of list in BinarySearchTree to be removed

| :return: None

|

| get(self, key)

| Same logic as put() method.

| :param key: index of list in BinarySearchTree

| :return: Returns payload, otherwise none/nothing else

|

| height(self, node)

| Height of tree root

| :param node: root node

| :return: height of tree

|

| items(self)

| :return:

|

| keys(self)

| Keys in list, or nodes in tree

| :return: None

|

| put(self, key, val)

| Builds Binary Search Tree.

| Checks to see if the tree has a root.

| If no root, creates a new TreeNode and installs it as root of tree.

| :param key: index of list in BinarySearchTree

| :param val: additional info contained in a node

| :return: None

|

| search\_ordered\_list(self, val)

| :param val:

| :return:

|

| search\_random\_list(self, val)

| :param val:

| :return:

|

| set\_balanceFactor(self)

| :return:

|

| values(self)

| :return:

|

| ----------------------------------------------------------------------

| Methods inherited from treeNode.TreeNode:

|

| hasAnyChildren(self)

| If a node has a lc node or rc node

| :return: either node for rc or node for lc

|

| hasBothChildren(self)

| If a node has both lc and rc nodes

| :return: both nodes for lc and rc

|

| hasLeftChild(self)

| The subtree containing the left child node

| :return: node for left child

|

| hasRightChild(self)

| The subtree containing the right child node

| :return: node for right child

|

| isLeaf(self)

| Node that has no children

| :return: does not return node for rc or node for lc

|

| isLeftChild(self)

| If node is the lc node

| :return: parent node and lc node attached to parent node

|

| isRightChild(self)

| If node is the rc node

| :return: parent node and rc node attached to parent node

|

| isRoot(self)

| (The root of the tree) node in tree with no incoming edges

| :return: does not return parent node

|

| replaceNodeData(self, key, value, lc, rc)

| Replaces original node data with new node data

| :param key: indices of list

| :param value: additional info contained in a node

| :param lc: node to the left below its parent node (incoming edges)-->node on left subtree

| :param rc: node to the right below its parent node (incoming edges)--> node on right subtree

| :return: None

|

| ----------------------------------------------------------------------

| Data descriptors inherited from treeNode.TreeNode:

|

| \_\_dict\_\_

| dictionary for instance variables (if defined)

|

| \_\_weakref\_\_

| list of weak references to the object (if defined)

>>>

testing.py

\_\_author\_\_ = **'Dennis Qiu'  
from** binarySearchTree **import** BinarySearchTree  
**from** AVLtree **import** AVLTree  
**import** time  
**import** sys  
**import** math  
  
predicted\_times = {}  
d\_sizes = [100, 1000, 10000,20000]  
tree\_dic = open(**'tree.txt'**, **'w'**, encoding=**'utf-8'**)  
  
**def** dictionary(dic\_file):  
 *"""  
 Reads in dictionary.txt.* **:param** *dic\_file: dictionary.txt file* **:return***: list of words in dictionary.txt  
 """* dic\_list = []  
 **with** open(dic\_file, **'r'**, encoding=**'UTF-8'**) **as** words:  
 **for** line **in** words.readlines():  
 dic = line.strip()  
 dic = dic.lower()  
 dic\_list.append(dic)  
 **return** dic\_list  
  
**def** O\_logn\_time(time1, size2, size1=d\_sizes[0]):  
 *"""* **:param** *time1:* **:param** *size2:* **:param** *size1:* **:return***: O logn time  
 """* **return** time1 \* math.log(size2) / math.log(size1)  
  
**def** O\_n\_time(time1, size2, size1=d\_sizes[0]):  
 *"""* **:param** *time1:* **:param** *size2:* **:param** *size1:* **:return***:  
 """* **return** time1 \* size2 / size1  
  
**def** predict\_bigO\_time(cur\_size, cur\_time, next\_size):  
 *"""  
 calculate the expected times for next size with different complexity performances  
 based on current time measured for current size  
 O(N2)  
 O(N)  
 O(NlogN)  
 O(logN)* **:param** *cur\_size:* **:param** *cur\_time:* **:param** *next\_size:* **:return***: None  
 """* t\_n2 = cur\_time \* next\_size \* next\_size / cur\_size / cur\_size  
 t\_n = cur\_time \* next\_size / cur\_size  
 t\_nlogn = cur\_time \* next\_size \* math.log(next\_size) / cur\_size / math.log(cur\_size)  
 t\_logn = cur\_time \* math.log(next\_size) / math.log(cur\_size)  
 print(**'Based on time measured for current size {}: {:5f} seconds, for next size {}:\n\  
 \tcurrent : {:5f}\n\  
 \tO(logN) : {:5f}\n\  
 \tO(N) : {:5f}\n\  
 \tO(NlogN) : {:5f}\n\  
 \tO(N2) : {:5f}\n'**.format(cur\_size, cur\_time, next\_size, cur\_time, t\_logn, t\_n, t\_nlogn, t\_n2))  
  
**def** search\_time(dic, d, Tree, orderedList=**False**):  
 *"""  
 Measures the time it takes to search word in the given list* **:param** *dic: the word (value of the node) to be searched* **:param** *d: the word list, it can be random or ordered list* **:param** *Tree: a tree object (can be BinarySearchTree or AVLTree)* **:param** *orderedList: a flag to indicate whether wList is an ordered list or not, by default is not* **:return***: measured time from start to finish  
 """* size = len(d)  
 **if** Tree == **"AVLTree"**:  
 tree = AVLTree()  
 **elif** Tree == **"BinarySearchTree"**:  
 tree = BinarySearchTree()  
 **else**:  
 print(**"Unsupported tree type: {}"**.format(Tree))  
 **return** 0  
 **if** orderedList:  
 list\_str = **'ordered list'  
 else**:  
 list\_str = **'random list'  
  
 try**:  
 **for** i **in** range(size):  
 tree[i] = d[i]  
 **except** RecursionError **as** e:  
 print(**'Exception caught while building the {} with size {}: {}'**.format(Tree, size, e))  
 tree\_dic.write(**'Error in building {}, size {:5d}, {}, word: "{}"\n'**.format(Tree, size, list\_str, dic))  
 **return** 0  
 print(**'{}, height of tree: {}, size of tree: {}'**.format(Tree, tree.height(tree.root), len(tree)))  
  
 tree.set\_balanceFactor()  
 start = time.clock()  
 **try**:  
 **if** orderedList:  
 dKey = tree.search\_ordered\_list(dic)  
 **else**:  
 dKey = tree.search\_random\_list(dic)  
 **if** dKey:  
 print(**'yes, {} is in tree, found at key {}'**.format(dic, dKey))  
 **else**:  
 print(**"no, {} is not found in tree"**.format(dic))  
 **except** RecursionError **as** e:  
 print(**'Exception caught during search with size {}: {}'**.format(size, e))  
 tree\_dic.write(**'Error during search {}, size {:5d}, {}, word: "{}"\n'**.format(Tree, size, list\_str, dic))  
 **return** 0  
 stop = time.clock()  
 d\_time = stop - start  
 print(**'Searching for word "{}" in size {} tree took {:5f} seconds'**.format(dic, size, d\_time))  
  
 **if** dKey:  
 word\_str = **'word in tree'  
 else**:  
 word\_str = **'word not in tree'** tree\_dic.write(**'\nSearch time: {:5f} seconds \n{}: size {}\n{}, word: "{}"\n'**.format(d\_time, list\_str, size, word\_str, dic))  
  
 sizes = d\_sizes  
 this\_size = sizes.index(size)  
 **if** this\_size < len(sizes) - 1:  
 nextSize = sizes[this\_size + 1]  
 predict\_bigO\_time(size, d\_time, nextSize)  
 **return** d\_time  
  
**def** main():  
 sys.setrecursionlimit(5000)  
  
 rand\_dic = dictionary(**'./dictionary.txt'**)  
 sort\_dic = sorted(rand\_dic)  
  
 rdic1 = rand\_dic[:100]  
 sdic1 = sort\_dic[:100]  
  
 r = rdic1[36]  
 s = sdic1[36]  
 not\_dic = **'dark'** tree\_dic.write(**'BinarySearchTree'**)  
 print(**'SEARCHING WORD IN TREE WITH RANDOM LIST'**)  
 ordered\_list = **False  
 with** open(**"./InRandomTree.csv"**, **"w"**, encoding=**'utf-8'**) **as** t:  
 t.write(**'size, BST, AVLT, O-logN, O-n\n'**)  
 **for** i **in** d\_sizes:  
 rdic = list(rand\_dic[:i])  
 bst = search\_time(r, rdic, **"BinarySearchTree"**, ordered\_list)  
 avlt = search\_time(r, rdic, **"AVLTree"**, ordered\_list)  
 **if** i == 100:  
 base\_t = avlt  
 t\_logn = base\_t  
 t\_n = base\_t  
 **else**:  
 t\_logn = O\_logn\_time(base\_t, i)  
 t\_n = O\_n\_time(base\_t, i)  
 t.write(**"{:5d}, {:6f}, {:6f}, {:6f}, {:6f}\n"**.format(i, bst, avlt, t\_logn, t\_n))  
 print(**'\t'**)  
  
 print(**'SEARCHING WORD NOT IN TREE WITH RANDOM LIST'**)  
 **with** open(**"./NotInRandomTree.csv"**, **"w"**, encoding=**'utf-8'**) **as** t:  
 t.write(**'size, BST, AVLT, O-logN, O-n\n'**)  
 **for** i **in** d\_sizes:  
 rdic = list(rand\_dic[:i])  
 bst = search\_time(not\_dic, rdic, **"BinarySearchTree"**, ordered\_list)  
 avlt = search\_time(not\_dic, rdic, **"AVLTree"**, ordered\_list)  
 **if** i == 100:  
 base\_t = avlt  
 t\_logn = base\_t  
 t\_n = base\_t  
 **else**:  
 t\_logn = O\_logn\_time(base\_t, i)  
 t\_n = O\_n\_time(base\_t, i)  
 t.write(**"{:5d}, {:6f}, {:6f}, {:6f}, {:6f}\n"**.format(i, bst, avlt, t\_logn, t\_n))  
 print(**'\t'**)  
  
 print(**'SEARCHING WORD IN TREE WITH ORDERED LIST'**)  
 ordered\_list = **True  
 with** open(**"./InOrderedTree.csv"**, **"w"**, encoding=**'utf-8'**) **as** t:  
 t.write(**'size, BST, AVLT, O-logN, O-n\n'**)  
 **for** i **in** d\_sizes:  
 sdic = list(sort\_dic[:i])  
 bst = search\_time(s, sdic, **"BinarySearchTree"**, ordered\_list)  
 avlt = search\_time(s, sdic, **"AVLTree"**, ordered\_list)  
 **if** i == 100:  
 base\_t = avlt  
 t\_logn = base\_t  
 t\_n = base\_t  
 **else**:  
 t\_logn = O\_logn\_time(base\_t, i)  
 t\_n = O\_n\_time(base\_t, i)  
 t.write(**"{:5d}, {:6f}, {:6f}, {:6f}, {:6f}\n"**.format(i, bst, avlt, t\_logn, t\_n))  
 print(**'\t'**)  
  
 print(**'SEARCHING WORD NOT IN TREE WITH ORDERED LIST'**)  
 **with** open(**"./NotInOrderedTree.csv"**, **"w"**, encoding=**'utf-8'**) **as** t:  
 t.write(**'size, BST, AVLT, O-logN, O-n\n'**)  
 **for** i **in** d\_sizes:  
 sdic = list(sort\_dic[:i])  
 bst = search\_time(not\_dic, sdic, **"BinarySearchTree"**, ordered\_list)  
 avlt = search\_time(not\_dic, sdic, **"AVLTree"**, ordered\_list)  
 **if** i == 100:  
 base\_t = avlt  
 t\_logn = base\_t  
 t\_n = base\_t  
 **else**:  
 t\_logn = O\_logn\_time(base\_t, i)  
 t\_n = O\_n\_time(base\_t, i)  
 t.write(**"{:5d}, {:6f}, {:6f}, {:6f}, {:6f}\n"**.format(i, bst, avlt, t\_logn, t\_n))  
 print(**'\t'**)  
 tree\_dic.close()  
**if** \_\_name\_\_ == **'\_\_main\_\_'**:  
 main()

tree.txt

BinarySearchTree  
Search time: 0.000086 seconds   
random list: size 100  
word in tree, word: "purflings"  
  
Search time: 0.000168 seconds   
random list: size 100  
word in tree, word: "purflings"  
  
Search time: 0.000119 seconds   
random list: size 1000  
word in tree, word: "purflings"  
  
Search time: 0.001460 seconds   
random list: size 1000  
word in tree, word: "purflings"  
Error in building BinarySearchTree, size 10000, random list, word: "purflings"  
  
Search time: 0.014516 seconds   
random list: size 10000  
word in tree, word: "purflings"  
Error in building BinarySearchTree, size 20000, random list, word: "purflings"  
  
Search time: 0.036127 seconds   
random list: size 20000  
word in tree, word: "purflings"  
  
Search time: 0.000175 seconds   
random list: size 100  
word not in tree, word: "dark"  
  
Search time: 0.000159 seconds   
random list: size 100  
word not in tree, word: "dark"  
  
Search time: 0.001747 seconds   
random list: size 1000  
word not in tree, word: "dark"  
  
Search time: 0.001467 seconds   
random list: size 1000  
word not in tree, word: "dark"  
Error in building BinarySearchTree, size 10000, random list, word: "dark"  
  
Search time: 0.015186 seconds   
random list: size 10000  
word not in tree, word: "dark"  
Error in building BinarySearchTree, size 20000, random list, word: "dark"  
  
Search time: 0.029050 seconds   
random list: size 20000  
word not in tree, word: "dark"  
  
Search time: 0.000197 seconds   
ordered list: size 100  
word in tree, word: "abampere"  
  
Search time: 0.000083 seconds   
ordered list: size 100  
word in tree, word: "abampere"  
  
Search time: 0.000121 seconds   
ordered list: size 1000  
word in tree, word: "abampere"  
  
Search time: 0.000071 seconds   
ordered list: size 1000  
word in tree, word: "abampere"  
Error in building BinarySearchTree, size 10000, ordered list, word: "abampere"  
  
Search time: 0.000083 seconds   
ordered list: size 10000  
word in tree, word: "abampere"  
Error in building BinarySearchTree, size 20000, ordered list, word: "abampere"  
  
Search time: 0.000105 seconds   
ordered list: size 20000  
word in tree, word: "abampere"  
  
Search time: 0.000193 seconds   
ordered list: size 100  
word not in tree, word: "dark"  
  
Search time: 0.000026 seconds   
ordered list: size 100  
word not in tree, word: "dark"  
  
Search time: 0.001607 seconds   
ordered list: size 1000  
word not in tree, word: "dark"  
  
Search time: 0.000073 seconds   
ordered list: size 1000  
word not in tree, word: "dark"  
Error in building BinarySearchTree, size 10000, ordered list, word: "dark"  
  
Search time: 0.000090 seconds   
ordered list: size 10000  
word not in tree, word: "dark"  
Error in building BinarySearchTree, size 20000, ordered list, word: "dark"  
  
Search time: 0.000100 seconds   
ordered list: size 20000  
word not in tree, word: "dark"

InRandomTree.csv

size, BST, AVLT, O-logN, O-n  
 100, 0.000086, 0.000168, 0.000168, 0.000168  
 1000, 0.000119, 0.001460, 0.000251, 0.001676  
10000, 0.000000, 0.014516, 0.000335, 0.016763  
20000, 0.000000, 0.036127, 0.000360, 0.033527

NotInRandomTree.csv

size, BST, AVLT, O-logN, O-n  
 100, , 0.00728, 0.008261, 0.008261 0.008261  
 1000, 0.013138, 0.011765, 0.012392, 0.082613  
10000, 0.000000, 0.051354, 0.016523, 0.826134  
20000, 0.000000, 0.060115, 0.017766, 1.652269

InOrderedTree.csv

size, BST, AVLT, O-logN, O-n  
 100, 0.000197, 0.000083, 0.000083, 0.000083  
 1000, 0.000121, 0.000071, 0.000125, 0.000832  
10000, 0.000000, 0.000083, 0.000166, 0.008325  
20000, 0.000000, 0.000105, 0.000179, 0.016649

NotInOrderedTree.csv

size, BST, AVLT, O-logN, O-n  
 100, 0.000193, 0.000026, 0.000026, 0.000026  
 1000, 0.001607, 0.000073, 0.000039, 0.000262  
10000, 0.000000, 0.000090, 0.000052, 0.002623  
20000, 0.000000, 0.000100, 0.000056, 0.005246