Author: Carlos Fernando Castaneda

Class: CS 2302

Date Modified: October 31, 2019

Instructor: Diego Aguirre

Assignment: Lab 4 B-Trees

TA: Gerardo Barraza

**Introduction:**

In python, b trees are a self-balancing data structure that takes the form of a tree and keeps data into a sorted format as well as keeping operations such as searches, insertions, and deletions in logarithmic time. In this lab, I will implement the functions of a tree to solve exactly the same problem from Lab 3, which was to import a text file called ‘Words.txt’ and find the number of anagrams there is of a certain word using the functions of the B-tree. The objective is not just to implement the b-tree to my existing code, but to also compare the three trees to see which is the best fastest to solve this problem.

**Proposed solution design and implementation:**

Before I begin, I want to note that the code from this lab is taken from the one in lab 3 since it is required in order to compare all the trees later in the results. I also want to note that some functions from the previous lab will be altered to meet the requirements of this lab.

There is not much to say about implementation of this lab, I mainly want to include the basic functions necessarily found when using b-trees so that I can traverse through the tree and use the same functions that the other trees got in the previous lab.

I also want to add a new section in the main file that will count the anagrams specifically for b trees. I suppose another search anagram method specifically for a b-tree would also be appropriate as it would require different steps from that of the normal search anagram method.

Lastly, I want to add another section on the opening menu where the user can access the B-tree as well as the AVL and Red and Black Trees as well. This will be very helpful to me later as it will help me determine the run times of the various trees included in my program.

**Experimental results**:

Before I begin explaining my results, like in the last lab, I wanted to note that I split my program into four separate files, one for the main method, one for the AVL Tree implementation, One for the Red and Black Implementation, and one for the B-tree implementation. I mainly had to do this due to the fact that the three trees have different Node functions, and I found that my implementation was simpler if I just split the trees into their own class, and I just imported them into the main program like how the professor did in our midterm 1 programs. I also don’t see any reason to explain these Tree classes as they are brought straight from Zybooks but have been altered so that it can be used in Python. With that out of the way, let’s begin!

The reference I got for the implementation for the B-tree I got from the code shared with us from the professor, which was from Professor Fuentes. I included most of the necessary methods that I found appropriate in this lab, such as search and split that will help me construct the b-tree necessary for the anagram search. Like the other trees, I decided to exclude any removal algorithms for the b-tree class as I found them redundant and unnecessary to the problem that I was trying to achieve.

The implementations for the search anagrams as well as the count anagrams was almost identical to the one I did in the previous lab, with the exception that I wrote specific methods for the b-tree implementation as well as creating a key list for searching for the amount of anagrams.

As mentioned above, I added a section in the user interface that will allow them to search for anagrams using all three trees from now on, although for when finding the second text file, it still searches for the anagrams using a red and black tree. This was made on purpose as that section of the lab does not really matter for the run time of the trees, so I decided to leave it on anyways. As for the run times, I have included the results at the end of the document before the conclusion.

Below you will find the results from various states in the program…

My results from running an AVL Tree, finding the word ‘weird’:

**A screenshot of a social media post

Description automatically generated**

My results from running a Red-Black Tree, finding the word ‘response’:

**A screenshot of a social media post

Description automatically generated**

My results from running a B-Tree, finding the word ‘pumpkin’:

**A screenshot of a social media post

Description automatically generated**

**Run times (Binary Search Trees):**

|  |  |  |
| --- | --- | --- |
| **Time** | **AVL Tree** | **Red Black Tree** |
| **1** | **23.94 seconds** | **24.82 seconds** |
| **2** | **24.28 seconds** | **26.02 seconds** |
| **3** | **24.47 seconds** | **23.54 seconds** |

**Run times (B-Tree):**

|  |  |  |  |
| --- | --- | --- | --- |
| **Time** | **1 key** | **5 keys** | **100 Keys** |
| **1** | **36.21 seconds** | **19.58 seconds** | **23.18 seconds** |
| **2** | **23.00 seconds** | **18.54 seconds** | **21.79 seconds** |
| **3** | **24.55 seconds** | **19.63 seconds** | **22.55 seconds** |

Comparing the times of all 3 trees shows us that the B-tree with the 5 keys is the fastest implementation of the three, and that is no surprise considering that the b tree takes logarithmic time to solve all its operations. The main reason I chose these specific keys to create the b-tree was to demonstrate the interval of times that can make the tree faster or slower, all depending on the value of the max key.

**Conclusions**:

With this lab, I was able to learn to code better using the Python language, including using multiple python files to create a b-tree implementation of the anagram problem introduced in Lab 3. I was also able to implement a way to import a text file and insert the strings inside of either an AVL, Red Black tree, and a B-tree and compare all three trees.

**Appendix :**

**Main.py**

*'''*

*Author: Carlos Fernando Castaneda*

*Class : CS 2302*

*Date Modified: October 29, 2019*

*Instructor: Diego Aguirre*

*Assingment: Lab 4 B Trees (Part 1 of 4)*

*TA: Gerardo Barraza*

*Purpose: To practice using B Trees to*

*solve the problem faced from Lab 3.*

*'''*

*#Used to calculate the time for each tree*

*import time*

*#Imports the AVL\_Trees and RB\_Trees programs to build the binary trees*

*import AVL\_Trees*

*import RB\_Trees*

*import B\_Trees*

*#Method that creates and returns an AVL tree from the imported file.*

*def AVL(filename):*

*AVL = AVL\_Trees.AVLTree()*

*key = 0*

*print("Preparing the AVL Tree now! Please wait...")*

*print()*

*with open(filename, encoding='windows-1252') as textFile:*

*for line in textFile:*

*string = line.split()*

*word = str(string)*

*stringedWord = word[2:len(word) - 2]*

*key = make\_key(stringedWord)*

*AVL.AVLTreeInsert(stringedWord, key)*

*return AVL*

*#Method that creates and returns an Red and Black tree from the imported file.*

*def RB(filename):*

*RB = RB\_Trees.RedBlack()*

*key = 0*

*print("Preparing the Red and Black Tree now! Please wait...")*

*print()*

*with open(filename, encoding='windows-1252') as textFile:*

*for line in textFile:*

*string = line.split()*

*word = str(string)*

*stringedWord = word[2:len(word) - 2]*

*key = make\_key(stringedWord)*

*RB.RBTreeInsert(stringedWord, key)*

*return RB*

*#Method that creates and returns an B tree from the imported file.*

*def Btree(filename):*

*print("How many keys do you want to implement on this B-Tree?")*

*key\_amount = int(input())*

*B\_tree = B\_Trees.BTree(key\_amount)*

*wordKey = ["", 0]*

*print("Preparing the B-Tree now! Please wait...")*

*print()*

*with open(filename, encoding='windows-1252') as textFile:*

*for line in textFile:*

*string = line.split()*

*word = str(string)*

*word\_str = word[2:len(word) - 2]*

*key = make\_key(word\_str)*

*wordKey = [word\_str, key]*

*B\_tree.Insert(wordKey)*

*return B\_tree*

*#Method that finds the necessary anagrams for both the maximum and count methods for the anagrams by using a binary search tree*

*def search\_anagrams(tree, word, prefix=""): #*

*if len(word) <= 1:*

*str = prefix + word*

*key = make\_key(str)*

*searched\_word = BSTsearch(tree, key)*

*if (searched\_word != None):*

*return 1*

*return 0*

*else:*

*count\_anagrams = 0*

*for i in range(len(word)):*

*current = word[i: i + 1]*

*before = word[0: i]*

*after = word[i + 1:]*

*if current not in before:*

*count\_anagrams += search\_anagrams(tree, before + after, prefix + current)*

*return count\_anagrams*

*#Method that finds the necessary anagrams for both the maximum and count methods for the anagrams by using a b-tree*

*def search\_anagrams\_btree(tree, word, prefix=""):*

*if len(word) <= 1:*

*str = prefix + word*

*key = make\_key(str)*

*key\_list = [str, key]*

*searched\_word = tree.Search(key\_list)*

*if (searched\_word != None):*

*return 1*

*return 0*

*else:*

*count\_anagrams = 0*

*for i in range(len(word)):*

*current = word[i: i + 1]*

*before = word[0: i]*

*after = word[i + 1:]*

*if current not in before:*

*count\_anagrams += search\_anagrams\_btree(tree, before + after, prefix + current)*

*return count\_anagrams*

*#Method that acts a binary search tree that will find the necessary anagrams for the find*

*def BSTsearch(tree, key):*

*cur = tree.root*

*while cur != None:*

*if key == cur.key:*

*return cur*

*elif key < cur.key:*

*cur = cur.left*

*else:*

*cur = cur.right*

*return None*

*#This method generates keys for each word by stringing together character numbers of each letter of a word.*

*def make\_key(word):*

*key = ""*

*res = list(word)*

*if (len(res[0]) > 1):*

*str\_word = str(res[0])*

*return make\_key(str\_word)*

*for i in range(len(res)):*

*char = caps(res[i])*

*char\_num = ord(char)*

*char\_str = str(char\_num)*

*key += char\_str*

*key\_int = int(key)*

*return key\_int*

*#Method activated by make key that will find the lower case letter by comparing it to its caps counterpart.*

*def caps(char):*

*cap\_list = ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'N', 'M', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z']*

*lower\_list = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'n', 'm', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z']*

*lower\_char = ''*

*for i in range(len(cap\_list)):*

*if (char == cap\_list[i]):*

*lower\_char = lower\_list[i]*

*return lower\_char*

*return char*

*#Method that counts the number of anagrams a certain word has.*

*def count\_anagrams(tree):*

*print("")*

*print("Please input a word for counting the number of anagrams: ")*

*new\_word = input()*

*count = search\_anagrams(tree, new\_word)*

*print("")*

*print("Findings complete! Total was:", count)*

*print()*

*#Method that counts the number of anagrams a certain word has using a Btree.*

*def count\_anagrams\_btree(tree):*

*print("")*

*print("Please input a word for counting the number of anagrams: ")*

*new\_word = input()*

*count = search\_anagrams\_btree(tree, new\_word)*

*print("")*

*print("Findings complete! Total was:", count)*

*print()*

*#Method that finds the maximum number of anagrams from another text file, the user can choose to ignore this part however.*

*def maximum\_anagrams():*

*print("Want to load another file to find the word that has the most anagrams?")*

*print("A. Yes")*

*print("B. No")*

*user\_selection = input()*

*count = 0*

*if (user\_selection == "A" or user\_selection == "a"):*

*print("")*

*print("Input filename of the textfile:")*

*filename = input()*

*tree = RB(filename)*

*anagrams = ""*

*anagrams\_num = 0*

*with open(filename, encoding='windows-1252') as textFile:*

*for line in textFile:*

*string = line.split()*

*word = str(string)*

*word\_str = word[2:len(word) - 2]*

*count = search\_anagrams(tree, word\_str)*

*if (anagrams\_num < count):*

*anagrams = word\_str*

*anagrams\_num = count*

*print("The word with the most anagrams is '", anagrams, "' with the total of", anagrams\_num, "anagrams.")*

*print("Terminating program, Farewell!")*

*elif (user\_selection == "B" or user\_selection == "b"):*

*print("User selected to exiting the program.")*

*print("Farewell.")*

*elif count == 3:*

*print("ERROR: Too many incorrect statements!")*

*print("Farewell.")*

*else:*

*print("ERROR: Input not valid!")*

*count+=1*

*maximum\_anagrams()*

*#Main method of the program, this method is used for the user to select the options for the rest of the program.*

*def main():*

*print("Welcome to the Anagram program! Please select the .txt file you want to implement:")*

*file = input()*

*print("")*

*print("Do you want to use an AVL tree or a Black-Red tree? Select from below:")*

*print("A. AVL Tree")*

*print("B. Red-Black Tree")*

*print("C. B-Tree")*

*user\_selection = input()*

*count = 0*

*if (user\_selection == 'A' or user\_selection == 'a'):*

*start1 = time.time()*

*tree = AVL(file)*

*end1 = time.time()*

*print("Tree insertion complete!")*

*print('Running time was: ', end1 - start1, 'seconds.')*

*count\_anagrams(tree)*

*elif (user\_selection == 'B' or user\_selection == 'b'):*

*start2 = time.time()*

*tree = RB(file)*

*end2 = time.time()*

*print("Tree insertion complete!")*

*print('Running time was: ', end2 - start2, 'seconds.')*

*count\_anagrams(tree)*

*elif (user\_selection == 'C' or user\_selection == 'c'):*

*start3 = time.time()*

*tree = Btree(file)*

*end3 = time.time()*

*print("Tree insertion complete!")*

*print('Running time was: ', end3 - start3, 'seconds.')*

*count\_anagrams\_btree(tree)*

*elif count == 3:*

*print("ERROR: Too many incorrect statements!")*

*print("Farewell.")*

*else:*

*print("ERROR: Input not valid!")*

*count+=1*

*main()*

*main()*

*maximum\_anagrams()*

**RB\_Trees.py**

*'''*

*Author: Carlos Fernando Castaneda*

*Class : CS 2302*

*Date Modified: October 29, 2019*

*Instructor: Diego Aguirre*

*Assingment: Lab 4 B Trees (Part 3 of 4)*

*TA: Gerardo Barraza*

*Purpose: To practice using B Trees to*

*solve the problem faced from Lab 3.*

*'''*

*class RBNode(object):*

*def \_\_init\_\_(self, word = "", key = 0, left = None, parent = None, right = None, color = -1):*

*self.word = word*

*self.key = 0*

*self.left = None*

*self.parent = None*

*self.right = None*

*self.color = -1*

*class RedBlack(object):*

*def \_\_init\_\_(self, root = None):*

*self.root = None*

*def RBTreeSetChild(self, parent, which\_child, child):*

*if (which\_child != "left" and which\_child != "right"):*

*return False*

*if (which\_child == "left"):*

*parent.left = child*

*else:*

*parent.right = child*

*if (child != None):*

*child.parent = parent*

*return True*

*def RBTreeReplaceChild(self, parent, current\_child, new\_child):*

*if (parent.left == current\_child):*

*return self.RBTreeSetChild(parent, "left", new\_child)*

*elif (parent.right == current\_child):*

*return self.RBTreeSetChild(parent, "right", new\_child)*

*return False*

*def RBTreeRotateLeft(self, node):*

*right\_left\_child = node.right.left*

*if (node.parent != None):*

*self.RBTreeReplaceChild(node.parent, node, node.right)*

*else:*

*self.root = node.right*

*self.root.parent = None*

*self.RBTreeSetChild(node.right, "left", node)*

*self.RBTreeSetChild(node, "right", right\_left\_child)*

*def RBTreeRotateRight(self, node):*

*left\_right\_child = node.left.right*

*if (node.parent != None):*

*self.RBTreeReplaceChild(node.parent, node, node.left)*

*else:*

*self.root = node.left*

*self.root.parent = None*

*self.RBTreeSetChild(node.left, "right", node)*

*self.RBTreeSetChild(node, "left", left\_right\_child)*

*def RBTreeInsert(self, words, key):*

*node = RBNode(words, key)*

*node.key = key*

*self.BSTinsert(node)*

*node.color = 1*

*self.RBTreeBalance(node)*

*def RBTreeGetGrandparent(self, node):*

*if (node.parent == None):*

*return None*

*return node.parent.parent*

*def RBTreeGetUncle(self, node):*

*grandparent = None*

*if (node.parent != None):*

*grandparent = node.parent.parent*

*if (grandparent == None):*

*return None*

*if (grandparent.left == node.parent):*

*return grandparent.right*

*else:*

*return grandparent.left*

*def RBTreeBalance(self, node):*

*if (node.parent == None):*

*node.color = 0*

*return*

*if (node.parent.color == 0):*

*return*

*parent = node.parent*

*grandparent = self.RBTreeGetGrandparent(node)*

*uncle = self.RBTreeGetUncle(node)*

*if (uncle != None and uncle.color == 1):*

*parent.color = uncle.color = 0*

*grandparent.color = 1*

*self.RBTreeBalance(grandparent)*

*return*

*if (node == parent.right and parent == grandparent.left):*

*self.RBTreeRotateLeft(parent)*

*node = parent*

*parent = node.parent*

*elif (node == parent.left and parent == grandparent.right):*

*self.RBTreeRotateRight(parent)*

*node = parent*

*parent = node.parent*

*parent.color = 0*

*grandparent.color = 1*

*if (node == parent.left):*

*self.RBTreeRotateRight(grandparent)*

*else:*

*self.RBTreeRotateLeft(grandparent)*

*def BSTinsert(self, node):*

*if (self.root == None):*

*self.root = node*

*node.left = None*

*node.right = None*

*else:*

*cur = self.root*

*while (cur != None):*

*if (node.key < cur.key):*

*if (cur.left == None):*

*cur.left = node*

*cur = None*

*else:*

*cur = cur.left*

*else:*

*if (cur.right == None):*

*cur.right = node*

*cur = None*

*else:*

*cur = cur.right*

*node.left = None*

*node.right = None*

**AVL\_Trees.py**

*'''*

*Author: Carlos Fernando Castaneda*

*Class : CS 2302*

*Date Modified: October 29, 2019*

*Instructor: Diego Aguirre*

*Assingment: Lab 4 B Trees (Part 2 of 4)*

*TA: Gerardo Barraza*

*Purpose: To practice using B Trees to*

*solve the problem faced from Lab 3.*

*'''*

*class AVLNode(object):*

*def \_\_init\_\_(self, word, key, left = None, parent = None, right = None):*

*self.word = word #Added attribute*

*self.key = 0*

*self.left = None*

*self.parent = None*

*self.right = None*

*self.height = -1*

*class AVLTree(object):*

*def \_\_init\_\_(self, root = None):*

*self.root = None*

*def AVLTreeUpdateHeight(self, node):*

*left\_height = -1*

*if (node.left != None):*

*left\_height = node.left.height*

*right\_height = -1*

*if (node.right != None):*

*right\_height = node.right.height*

*node.height = max(left\_height, right\_height) + 1*

*def AVLTreeSetChild(self, parent, which\_child, child):*

*if (which\_child != "left" and which\_child != "right"):*

*return False*

*if (which\_child == "left"):*

*parent.left = child*

*else:*

*parent.right = child*

*if (child != None):*

*child.parent = parent*

*self.AVLTreeUpdateHeight(parent)*

*return True*

*def AVLTreeReplaceChild(self, parent, curr\_child, new\_child):*

*if (parent.left == curr\_child):*

*return self.AVLTreeSetChild(parent, "left", new\_child)*

*elif (parent.right == curr\_child):*

*return self.AVLTreeSetChild(parent, "right", new\_child)*

*return False*

*def AVLTreeGetBalance(self, node):*

*left\_height = -1*

*if (node.left != None):*

*left\_height = node.left.height*

*right\_height = -1*

*if (node.right != None):*

*right\_height = node.right.height*

*return left\_height - right\_height*

*def AVLTreeRotateRight(self, node):*

*left\_right\_child = node.left.right*

*if (node.parent != None):*

*self.AVLTreeReplaceChild(node.parent, node, node.left)*

*else: # node is root*

*self.root = node.left*

*self.root.parent = None*

*self.AVLTreeSetChild(node.left, "right", node)*

*self.AVLTreeSetChild(node, "left", left\_right\_child)*

*def AVLTreeRotateLeft(self, node):*

*right\_left\_child = node.right.left*

*if (node.parent != None):*

*self.AVLTreeReplaceChild(node.parent, node, node.right)*

*else:*

*self.root = node.right*

*self.root.parent = None*

*self.AVLTreeSetChild(node.right, "left", node)*

*self.AVLTreeSetChild(node, "right", right\_left\_child)*

*def AVLTreeRebalance(self, node):*

*self.AVLTreeUpdateHeight(node)*

*if (self.AVLTreeGetBalance(node) == -2):*

*if (self.AVLTreeGetBalance(node.right) == 1):*

*self.AVLTreeRotateRight(node.right)*

*return self.AVLTreeRotateLeft(node)*

*elif (self.AVLTreeGetBalance(node) == 2):*

*if (self.AVLTreeGetBalance(node.left) == -1):*

*self.AVLTreeRotateLeft(node.left)*

*return self.AVLTreeRotateRight(node)*

*return node*

*def AVLTreeInsert(self, words, key):*

*node = AVLNode(words, key)*

*node.key = key*

*if (self.root == None):*

*self.root = node*

*node.parent = None*

*return*

*cur = self.root*

*while (cur != None):*

*if (node.key < cur.key):*

*if (cur.left == None):*

*cur.left = node*

*node.parent = cur*

*cur = None*

*else:*

*cur = cur.left*

*else:*

*if (cur.right == None):*

*cur.right = node*

*node.parent = cur*

*cur = None*

*else:*

*cur = cur.right*

*node = node.parent*

*while (node != None):*

*self.AVLTreeRebalance(node)*

*node = node.parent*

**B\_Trees.py**

*'''*

*Author: Carlos Fernando Castaneda*

*Class : CS 2302*

*Date Modified: October 29, 2019*

*Instructor: Diego Aguirre*

*Assingment: Lab 4 B Trees (Part 4 of 4)*

*TA: Gerardo Barraza*

*Purpose: To practice using B Trees to*

*solve the problem faced from Lab 3.*

*'''*

*# This class in the program is used to create objects of BTrees, or binary trees*

*class BTreeNode:*

*# Constructor*

*def \_\_init\_\_(self, item=[], child=[], is\_leaf=True, max\_items=5):*

*self.keys = item*

*self.child = child*

*self.is\_leaf = is\_leaf*

*#If the max item will equal 3 if it is less than 3*

*if max\_items < 3:*

*max\_items = 3*

*#If the max items is not odd or greater than 3, it will change it so that it must be odd and greater or equal to 3*

*if max\_items % 2 == 0:*

*max\_items += 1*

*self.max\_items = max\_items*

*#Will return if the leaves are full or not*

*def is\_full(self):*

*return len(self.keys) >= self.max\_items*

*#Method that acts as the operator of the B Tree operations*

*class BTree:*

*#Creates the Constructor*

*def \_\_init\_\_(self, max\_items=5):*

*self.max\_items = max\_items*

*self.root = BTreeNode(max\_items=max\_items)*

*# Method that is used to find the correct index position of child*

*def FindChild(self, k, node=None):*

*if node is None:*

*node = self.root*

*key\_num = 0*

*for i in range(len(node.keys)):*

*key = node.keys[i]*

*key\_num = k[1]*

*if key\_num < key[1]:*

*return i*

*return len(node.keys)*

*#Method that is used to insert items into the binary tree into non-leaf nodes*

*def InsertInternal(self, i, node=None):*

*if node is None:*

*node = self.root*

*if node.is\_leaf:*

*self.InsertLeaf(i, node)*

*else:*

*k = self.FindChild(i, node)*

*if node.child[k].is\_full():*

*m, l, r = self.split(node.child[k])*

*node.keys.insert(k, m)*

*node.child[k] = l*

*node.child.insert(k + 1, r)*

*k = self.FindChild(i, node)*

*self.InsertInternal(i, node.child[k])*

*# Method is used to split full nodes to be used throughout the tree*

*def split(self, node=None):*

*if node is None:*

*node = self.root*

*mid = node.max\_items // 2*

*if node.is\_leaf:*

*leftChild = BTreeNode(node.keys[:mid], max\_items=node.max\_items)*

*rightChild = BTreeNode(node.keys[mid + 1:], max\_items=node.max\_items)*

*else:*

*leftChild = BTreeNode(node.keys[:mid], node.child[:mid + 1], node.is\_leaf, max\_items=node.max\_items)*

*rightChild = BTreeNode(node.keys[mid + 1:], node.child[mid + 1:], node.is\_leaf, max\_items=node.max\_items)*

*return node.keys[mid], leftChild, rightChild*

*# Method that is used to get number of leaves that are full in the tree*

*def InsertLeaf(self, i, node=None):*

*if node is None:*

*node = self.root*

*node.keys.append(i)*

*node.keys.sort(key = lambda r:r[1])*

*# Method that is used to get number of leaves that are full in the tree*

*def leaves(self, node=None):*

*if node is None:*

*node = self.root*

*if node.is\_leaf:*

*return [node.keys]*

*s = []*

*for c in node.child:*

*s = s + self.leaves(c)*

*return s*

*# Method is used to insert items into the nodes of the tree*

*def Insert(self, i, node=None):*

*if node is None:*

*node = self.root*

*if not node.is\_full():*

*self.InsertInternal(i, node)*

*else:*

*m, l, r = self.split(node)*

*node.keys = [m]*

*node.child = [l, r]*

*node.is\_leaf = False*

*k = self.FindChild(i, node)*

*self.InsertInternal(i, node.child[k])*

*# Method that is used to find height of tree*

*def height(self, node=None):*

*if node is None:*

*node = self.root*

*if node.is\_leaf:*

*return 0*

*return 1 + self.height(node.child[0])*

*# Method is used to search item in a B Tree*

*def Search(self, k, node=None):*

*if node is None:*

*node = self.root*

*for i in range(len(node.keys)):*

*key = node.keys[i]*

*if (k[1] == key[1]):*

*return node*

*if node.is\_leaf:*

*return None*

*return self.Search(k, node.child[self.FindChild(k, node)])*

I certify that this project is entirely my own work. I wrote, debugged, and tested the code being presented, performed the experiments, and wrote the report. I also certify that I did not share my code or report or provide inappropriate assistance to any student in the class.

