CS 2302

Lab 1 Report

Author: David Ayala

ID: 80617838

Instructor: Diego Aguirre

TA: Gerardo Barraza

**Introduction**

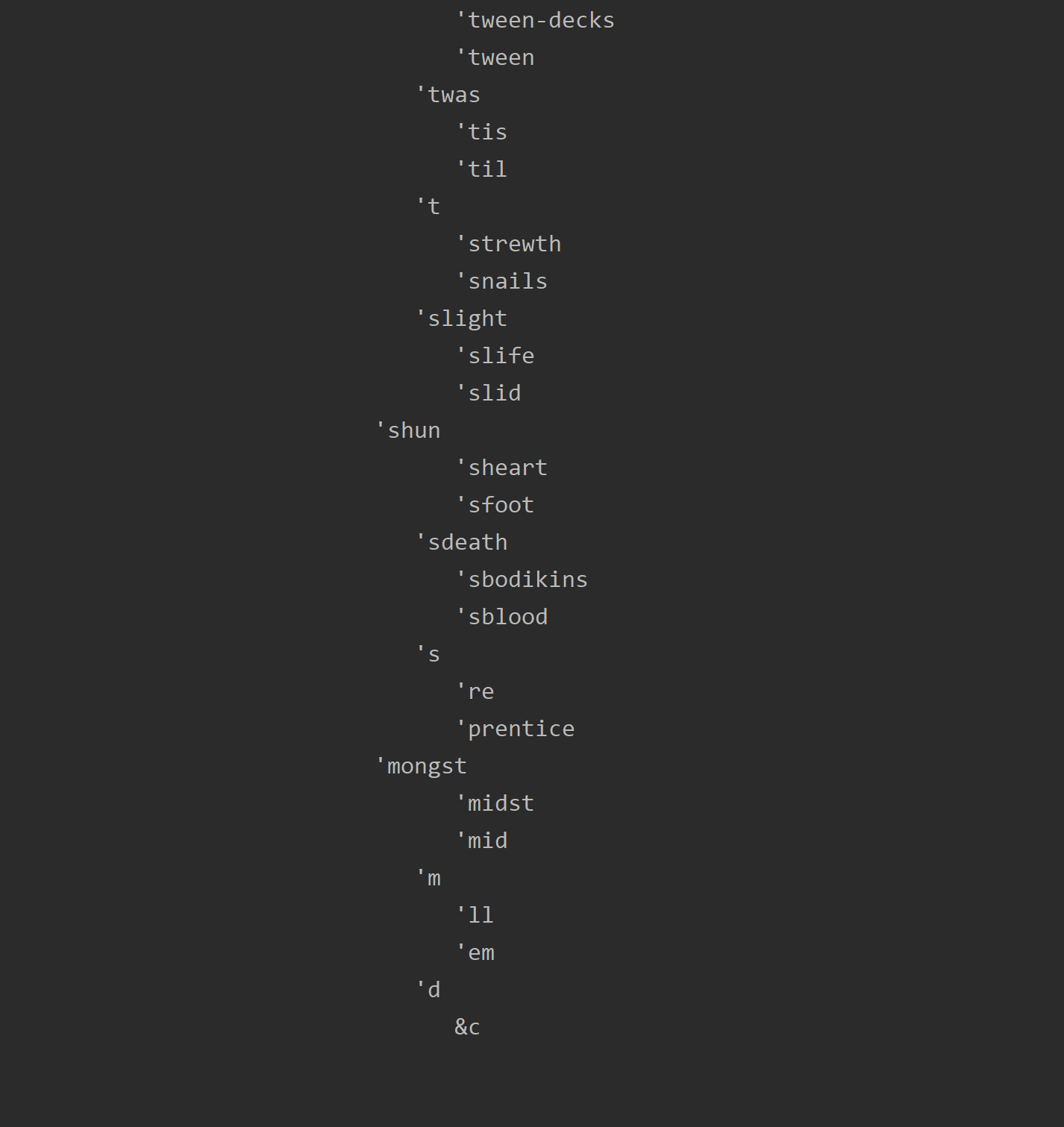
The purpose of this lab was to do the same as in Lab 3, where we used binary trees to solve an interesting anagrams problem. but in this lab, we are asked to solve the same problem using B-Trees. So please refer back to lab 3 for more details.

**Proposed Solution & Design Implementation**

So I understood the basic idea about how to do it, but I was unable accomplish this due to me being unable to make my search function work. But I was able to get the be=Tree implemented but ultimately I keep trying how this lab until I had to finally just submit it

**Experimental Results**

I didn’t put the full picture of the Btree cause it was really big but I think you understand it.



**Conclusion**

I tried to this lab for long periods of time but I also had to study for exams this week, so my time was very split between this lab and exams. I think that both things suffered and that I need to manage my time better so that I will be able to accomplish both next time. The bottom line is that I really tried to do this lab but I was unable to accomplish it so I will need to review zybooks and other resources that I may be able to this lab again later for my own improvement.

**Appendix**

# code is Olac Fuentes' from last semester when I had him with few changes

class BTreeNode:

# Constructor

def \_\_init\_\_(self, keys=[], children=[], isLeaf=True, maxKeys=5):

self.keys = keys

self.children = children

self.isLeaf = isLeaf

if maxKeys < 3: # maxKeys must be odd and greater or equal to 3

maxKeys = 3

if maxKeys % 2 == 0: # maxKeys must be odd and greater or equal to 3

maxKeys += 1

self.maxKeys = maxKeys

def is\_full(self):

return len(self.keys) >= self.maxKeys

class BTree:

def \_\_init\_\_(self, maxKeys=5):

self.maxKeys = maxKeys

self.root = BTreeNode(maxKeys=maxKeys)

def find\_child(self, k, node=None):

if node is None:

node = self.root

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

if k < node.keys[i]:

return i

return len(node.keys)

def insert\_internal(self, i, node=None):

if node is None:

node = self.root

# node cannot be Full

if node.isLeaf:

self.insert\_leaf(i, node)

else:

k = self.find\_child(i, node)

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

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

node.keys.insert(k, m)

node.children[k] = l

node.children.insert(k + 1, r)

k = self.find\_child(i, node)

self.insert\_internal(i, node.children[k])

def split(self, node=None):

if node is None:

node = self.root

mid = node.maxKeys // 2

if node.isLeaf:

left\_child = BTreeNode(node.keys[:mid], maxKeys=node.maxKeys)

right\_child = BTreeNode(node.keys[mid + 1:], maxKeys=node.maxKeys)

else:

left\_child = BTreeNode(node.keys[:mid], node.children[:mid + 1], node.isLeaf,

maxKeys=node.maxKeys)

right\_child = BTreeNode(node.keys[mid + 1:], node.children[mid + 1:], node.isLeaf,

maxKeys=node.maxKeys)

return node.keys[mid], left\_child, right\_child

def insert\_leaf(self, i, node=None):

if node is None:

node = self.root

node.keys.append(i)

node.keys.sort()

def leaves(self, node=None):

if node is None:

node = self.root

if node.isLeaf:

return [node.keys]

s = []

for c in node.children:

s = s + self.leaves(c)

return s

def insert(self, i, node=None):

if node is None:

node = self.root

if not node.is\_full():

self.insert\_internal(i, node)

else:

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

node.keys = [m]

node.children = [l, r]

node.isLeaf = False

k = self.find\_child(i, node)

self.insert\_internal(i, node.children[k])

def height(self, node=None):

if node is None:

node = self.root

if node.isLeaf:

return 0

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

def contains(self, k):

result = self.search(k)

if result is None:

return False

return True

def search(self, k, node=None):

if node is None:

node = self.root

if k in node.keys:

return node

if node.isLeaf:

return None

return self.search(k, node.children[self.find\_child(k, node)])

def print\_d(self, space, node=None):

if node is None:

node = self.root

if node.isLeaf:

for i in range(len(node.keys) - 1, -1, -1):

print(space, node.keys[i])

else:

self.print\_d(space + ' ', node.children[len(node.keys)])

for i in range(len(node.keys) - 1, -1, -1):

print(space, node.keys[i])

self.print\_d(space + ' ', node.children[i])