Joaquin Hidalgo

CS 2302 MW @1:30pm

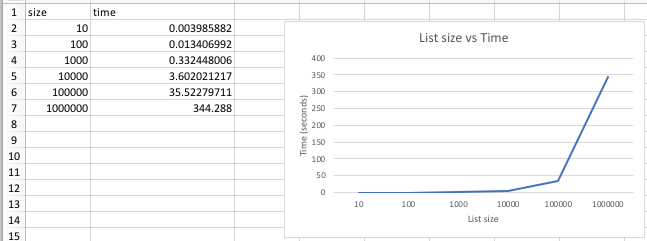
Lab 04

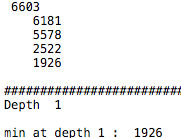
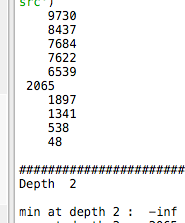
**B-Tree’s**

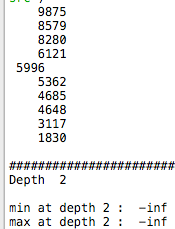
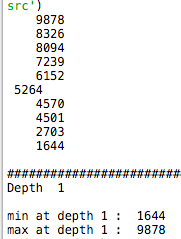
**Introduction:** The problem at hand is to demonstrate how to traverse through a B-Tree. As well as write methods within a B-Tree such as, finding the depth of a certain item or finding the minimum item from the B-Tree in the tree and so on. Lastly, to record and look at the time my program took to run with different list size’s.

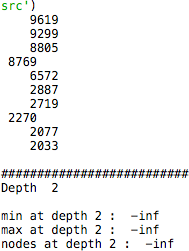
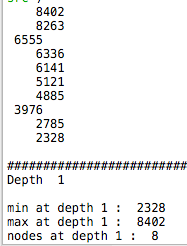
**Proposed solution design and implementation:** To show the height of the Tree, I would just need to go down each child tree until I am at a leaf node. Once at a leaf node return the depth of the item and that is the height of the tree because all B-Trees are balanced. To extract the B-Tree into a sorted list, I would like to start with the minimum item in the tree and append to list as traverse through the B-Tree. To return the min or max item of the Tree, I would need to go to the child position index of 0 each time I am at a leaf node, then return item at index 0 for minimum. To return max do the same just enter each child tree node at index -1 and once at leaf node, return the item in position -1. To return the number of items in a Tree is to return the summation of the length of item of each node in the B-Tree, while visiting each node, sum up the length of the items in the B-Tree. To count the number of full nodes in the tree, while traversing through the tree, if the length of items matches the max number of items in the Tree, increase counter by one and return once at the end of the tree. Lastly, to return the depth of an item is simply traversing through the B-Tree by comparing the searchee item to the items in the B-Tree. That way you do not have to traverse through the whole Tree. Once at the end either return the summation of depths you took to arrive at the item or if you are at a leaf node and could not find item, then return -1.

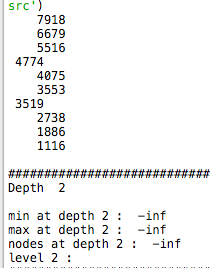
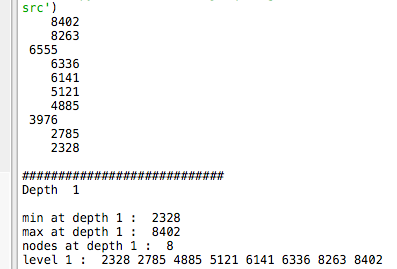
**Experimental results:**

Time vs List size 🡪 ****

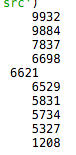
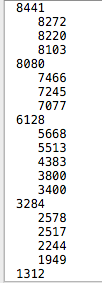
Min at depth 🡪  ** **

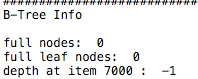
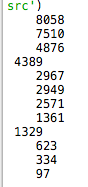
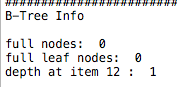
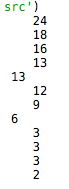
Max at depth 🡪 **** ****

Number of Nodes 🡪  ** **

Print items at Depth 🡪  ** **

Number Full nodes at Leaf && Number of Full Nodes 🡪

 || 

Depth at item 🡪 **** || ****

**Conclusion:** I learned how to search through a B-Tree. I learned different ways to traverse through the Tree. Also, I learned about depth and height of a tree. The height is the number of edges it takes to get from root to leaf node. Lastly, I learned to traverse the B-Tree in a way that I can add all the smaller items in the tree first to the list, thus making a sorted list.

**Code:**

﻿#!/usr/bin/env python3

# -\*- coding: utf-8 -\*-

"""

Created Sun 10 Mar 21:39:32

Instructor: Olac Fuentes

TA: Dita and Mali

Last modified Mon 18 Mar 14:22:20

Implement B-Trees functions to further understand different the structure of BTree.

Understand how to use iterative and recursive methods to traverse BTree.

Display how long the program took to run in seconds.

"""

import math

import random

import time

class BTree(object):

# Constructor

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

self.item = item

self.child = child

self.isLeaf = isLeaf

if max\_items <3: #max\_items must be odd and greater or equal to 3

max\_items = 3

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

max\_items +=1

self.max\_items = max\_items

#Gives you index of appropiate child to better search for k

def FindChild(T,k):

# Determines value of c, such that k must be in subtree T.child[c], if k is in the BTree

for i in range(len(T.item)):

if k < T.item[i]:

return i

return len(T.item)

def InsertInternal(T,i):

# T cannot be Full

if T.isLeaf:

InsertLeaf(T,i)

else:

k = FindChild(T,i)

if IsFull(T.child[k]):

m, l, r = Split(T.child[k])

T.item.insert(k,m)

T.child[k] = l

T.child.insert(k+1,r)

k = FindChild(T,i)

InsertInternal(T.child[k],i)

#split full node

def Split(T):

#print('Splitting')

#PrintNode(T)

mid = T.max\_items//2

if T.isLeaf:

leftChild = BTree(T.item[:mid])

rightChild = BTree(T.item[mid+1:])

else:

leftChild = BTree(T.item[:mid],T.child[:mid+1],T.isLeaf)

rightChild = BTree(T.item[mid+1:],T.child[mid+1:],T.isLeaf)

return T.item[mid], leftChild, rightChild

def InsertLeaf(T,i):

T.item.append(i)

T.item.sort()

#returns boolean if node is full

def IsFull(T):

return len(T.item) >= T.max\_items

#insert into btree

def Insert(T,i):

if not IsFull(T):

InsertInternal(T,i)

else:

m, l, r = Split(T)

T.item =[m]

T.child = [l,r]

T.isLeaf = False

k = FindChild(T,i)

InsertInternal(T.child[k],i)

#Returns height of tree

def height(T):

if T.isLeaf:

return 0

return 1 + height(T.child[0])

#Returns node where k is, or None if k is not in the tree

def Search(T,k):

if k in T.item:

return T

if T.isLeaf:

return None

return Search(T.child[FindChild(T,k)],k)

# Prints items in tree in ascending order

def Print(T):

if T.isLeaf:

for t in T.item:

print(t,end=' ')

else:

for i in range(len(T.item)):

Print(T.child[i])

print(T.item[i],end=' ')

Print(T.child[len(T.item)])

# Prints items and structure of B-tree

def PrintD(T,space):

if T.isLeaf:

for i in range(len(T.item)-1,-1,-1):

print(space,T.item[i])

else:

PrintD(T.child[len(T.item)],space+' ')

for i in range(len(T.item)-1,-1,-1):

print(space,T.item[i])

PrintD(T.child[i],space+' ')

#Search and print item

def SearchAndPrint(T,k):

node = Search(T,k)

if node is None:

print(k,'not found')

else:

print(k,'found',end=' ')

print('node contents:',node.item)

#Recursivly iterate thru node and return height of tree

def Height(T):

#base case

if T.isLeaf:

return 0

return 1 + Height(T.child[0])

#Extract B-Tree in sorted order to array

def ExtractToSort(T,Z):

#base case

if T.isLeaf:

for i in range(len(T.item)):

Z.append(T.item[i] )

else:

for j in range(len(T.child)):

ExtractToSort(T.child[j], Z)

#prevents out of bounds by not adding the index of greatest T.child

if j != len(T.item):

Z.append(T.item[j] )

def Minimum(T, d):

if d == 0:

return T.item[0]

if T.isLeaf:

return -math.inf

#return minimum array(furthest left)

else:

return Minimum(T.child[0], d-1)

#Returns item in the furthest bottom right poition(largest)

def Maximum(T, d):

if d == 0:

return T.item[-1]

if T.isLeaf:

return -math.inf

#return maximum array(furthest right)

else:

return Maximum(T.child[-1], d-1)

#once at desired depth 'd', return the lengh of that item[] and return that

def NumNodesAtDepth(T,d):

if d ==0:

return len(T.item)

if T.isLeaf:

return -math.inf

#Not at desired depth 'd', recursivly call and decrease d by 1

else:

s = 0

for i in range(len(T.child)):

s+= NumNodesAtDepth(T.child[i],d-1)

return s

#once at desired depth 'd', print the items of that array and return that

def PrintByLevel(T, d):

if d == 0:

for i in range(len(T.item)):

print(T.item[i], end= ' ')

if T.isLeaf:

return

#Recursive call the next depth of nodes

else:

for i in range(len(T.child)):

PrintByLevel(T.child[i], d-1)

#check if node full while recursivly traversing each node

def NumOfFullNodes(T):

if T.isLeaf:

if NumNodesAtDepth(T,0) == T.max\_items:

return 1

return 0

#Traverse each child node

else:

s =0

for i in range(len(T.child)):

s+= NumOfFullNodes(T.child[i])

if len(T.item) == T.max\_items:

return 1 +s

return s

#When checking every leaf node, return the number of full leaf nodes

def NumOfFullLeafs(T):

if T.isLeaf:

if NumNodesAtDepth(T,0) == T.max\_items:

return 1

else:

return 0

#Traverse through each child tree

else:

s = 0

for i in range(len(T.child)):

s+= NumOfFullLeafs(T.child[i])

return s

#Traverse to 'k' and either return -1 if not found or return depth of 'k'

def DepthAtNode(T, k):

#Check root node

temp = T

if k in temp.item:

return 0

if temp.isLeaf:

return -1

count = 0

#check others nodes than root

while not temp.isLeaf:

temp = temp.child[FindChild(temp,k)]

count += 1

if k in temp.item:

return count

#If k NOT found

if temp.isLeaf:

return -1

##############################################

#U.I.

depth = 1

findThisItem = 12

L = []

T = BTree()

A = []

G = []

##########################

#Fill List

for i in range(12):

L.append(random.randint(0,25) )

#Fill the B-Tree

for i in L:

Insert(T,i)

startTime = time.time()

PrintD(T,'')

print()

print("############################")

print("Depth ", depth)

print()

print("min at depth", depth, ": ", Minimum(T, depth) )

print("max at depth", depth, ": ", Maximum(T,depth) )

print("nodes at depth",depth, ": ", NumNodesAtDepth(T,depth) )

print("level", depth, ": " , end = " ")

PrintByLevel(T, depth)

print()

print("############################")

print("B-Tree Info")

print()

print("full nodes: ", NumOfFullNodes(T) )

print("full leaf nodes: ", NumOfFullLeafs(T) )

print("depth at item", findThisItem, ": ", DepthAtNode(T,findThisItem) )

print()

ExtractToSort(T, G)

print("extracted list: ", end = ' ')

print(G)

endTime= time.time()

print("program time: ", endTime-startTime)

**Standards of Conduct and Academic Dishonesty**

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 provided inappropriate assistance to any student in the class.