Maria F. Corona Ortega
ID: 80560734
E-Mail: mfcoronaortega@miners.utep.edu
CS 2302 | MW 1:30-2:50 am
Instructor: Dr. Olac Fuaentes
TA: Anindita Nath

CS 2302 Lab 4 Report

Introduction

In this Lab we are given a BTree with max size of five which means each node contains five elements. These elements can contain a total of five elements each. All elements are sorted and we experiment with different basic operations. We are to implement the different operations on this given source code that already has operations to create BTree along with other basic operations.

Proposed Solution

- **Task 1 (Compute the height of the tree):** The approach to this task is recursive. Every time we traverse from child to child we count and return.
- **Task 2 (Extract the items in the B-tree into a sorted list):** This method is meant to be very similar to the print in order method. However rather than printing the BTree in ascending order, it will append elements in order into a list.
- **Task 3 (Return the minimum element in the tree at a given depth d):** Recursively we traverse to the far end of the BTree at given depth and return the element to the farthest left of such depth. Due to the structure of the BTree and how it is built, it will return the smallest element.
- **Task 4 (Return the maximum element in the tree at a given depth d):** similarly to the previous method, the tree is traversed recursively as well. In this method however we are going to the far right of the tree at whatever method and again due to the nature of the structure that will be the highest value at such depth.
- **Task 5 (Return the number of nodes in the tree at a given depth d):** Using a counter we traverse the list only up to given depth. The recursion is within a loop that will ensure we traverse everyone and count it returning the number of nodes in the tree and whatever depth.
- **Task 6 (Print all the items in the tree at a given depth d):** This like the previous method will have a similar structure. We will traverse the tree up to given depth and at such depth we will print all items with a combination of a loop and recursion.
- **Task 7 (Return the number of nodes in the tree that are full):** This method will traverse the tree and compare the length of every node to the preset maximum length of nodes. If its full returns 1. We keep count similarly with a for loop and a counter with a recursive call for all children.
- **Task 8 (Return the number of leaves in the tree that are full):** This method similarly to the previous, will traverse the tree and compare the length of every node to the preset maximum length of nodes. Since we need full leaves not nodes,

we first check if the current node is a leaf and then compare and return 1. We keep count similarly with a for loop and a counter with a recursive call for all children.
- **Task 9 (Given a key k, return the depth at which it is found in the tree, of -1 if k is not in the tree):** We traverse the tree and check is k is found within the node while keeping a counter for every level we traverse. If such integer is not present we return -1.

Methods

- ***CompHeight*:** This method takes in a single parameter this being a BTree. We first check if T.isLeaf meaning if the tree has a single node or is empty, if this is the case the height returns 0. If this is not the case we return 1 and recursively call this method at T.child at 0. This will return the height of the tree accurately due to the nature of the structure. The height is uniform throughout so even if we used T.child at 1, results will yield the same.
- ***ToList*:** This method receives two parameters one being the BTree *T* and the other being the empty list *L* to which we append the elements to. We then check if T is leaf, is so we append all items in T to the list. If T is not a leaf we traverse all children of T recursively until we hit a leaf then append. Once this is done we recursively call with the rest of the children. Lastly we return *L* a list which contains all elements in *T* in order.
- ***MinAtDepth*:** We receive two parameters the BTree *T* and the depth we need the minimum at, *d.* If the depth we need is 0 we simply return the first element in current item. If the depth is not 0 we recursively call the method which traverses to the first child of current item until we reach the needed depth by decreasing *d* by 1. once there we return the first element on current item which will be the smallest number.
- ***MaxAtDepth*:** We receive two parameters the BTree *T* and the depth we need the maximum at, *d.* If the depth we need is 0 we simply return the last element in current item. If the depth is not 0 we recursively call the method which traverses to the last child of current item until we reach the needed depth by decreasing *d* by 1. once there we return the last element on current item which will be the largest number.
- ***NodesAtDepth*:** In this method we receive two parameters, the BTreee *T* and the depth at which to count the nodes, *d.* We first check if the depth we need is 0, if so we simply return 1, the root will only contain a single node. We then check if the T is a Leaf but now checking for a depth over 0, if so we return the length of such child. Otherwise we transverse a for loop for all elements in T.child, and into a variable preset at count = 0, we add and recursively call the method for all items in T.child and decrease *d* by 1. When we reach *d*=0 we should have a count of the nodes which we then return.
- ***PrintAtDepth*:** This method like the previous two, receive parameters *T* and *d.* If the desired depth is 0, we print all elements in current item. Else we traverse all

children and recursively call the method for every element in T.child and decrease *d* by 1. When reaching the desired depth in tree, we print all elements at such depth per child.

- ***FullNodes*****:** This method only receives parameter *T*. We initiate counter and we traverse the tree. If T is none we check if the nodes in T are full by comparing to *T.max_items.* If so we return 1. We then check if *T* is a leaf, if so we also compare the items to see if the node is full, if so we also return 1. Lastly we a for loop we traverse all children of T and recursively call the same method into a counter for all children in T. Once everything is traversed and checked we return the count of full nodes. <u>The implementation of method does not yield the correct result.</u>
- ***FullLeaves*****:** This method receives a single parameter T. We first check if T is a leaf, is so, we check we check the size to see if its full. If the current node is full we return 1, else return 0. If T is not a leaf we traverse all children until we see a leaf into a counter. The purpose of this is to check all leafs and count the ones that are full. Lastly we return the count.
- ***FindDepth*****:** We receiver two parameters, one is the tree *T*, the other is the item we are looking for *k*. We first check if T is empty, if so we return -1. We then check if T is a leaf, if so we check if *k* is found in the current item while returning either 1 or 0. We call into a counter *depth* and return *depth.* <u>The implementation of method does not yield the correct result.</u>

Experimental Results

For the implementation of this we generate a list of random integers ranging from 0 to the size of the list duplicated. All methods are tested with the same Binary search tree per size. When size is increase, the integers in the tree do change but are kept constant through testing.

CompHeight:

N elements	Running Time
10	2.4324021069332957e-05
25	2.4348992155864835e-05
50	4.4419983169063926e-05
100	3.574899164959788e-05

ToList**:**

N elements	Running Time
10	6.671005394309759e-06
25	1.7606012988835573e-05
50	2.732500433921814e-05
100	5.386199336498976e-05

MinAtDepth:

N Elements	d = 0	d = 1	d = 2	d=3
10	8.811699808575213e-05	8.813702152110636e-05	N/A	N/A
25	9.652000153437257e-05	0.00016810200759209692	0.00010766400373540819	N/A
50	0.00015147801605053246	0.00011699998867698014	0.00011740901391021907	N/A
100	0.0001776489953044802	0.00011597300181165338	0.00011554901720955968	0.00011630900553427637

MaxAtDepth:

N Elements	d = 0	d = 1	d = 2	d=3
10	8.71980155352503e-05	8.826900739222765e-05	N/A	N/A
25	8.808498387224972e-05	0.00016755500109866261	0.00017042201943695545	N/A
50	0.00011741399066522717	0.0001180509862024337	0.001206307002576068	N/A
100	0.0001879379851743579	0.00011948900646530092	0.00011616499978117645	0.0001198059762828052

NodesAtDepth:

N Elements	d = 0	d = 1	d = 2	d=3
10	8.85640038177371e-05	0.00010634399950504303	N/A	N/A
25	0.00015458199777640402	9.712600149214268e-05	0.0001476079924032092	N/A

N Elements	d = 0	d = 1	d = 2	d=3
50	0.00011629398795776069	0.00011803899542428553	0.00012281598174013197	N/A
100	0.00011729099787771702	0.00016384999617002904	0.00011947599705308676	0.00012869300553575158

PrintAtDepth:

N Elements	d = 0	d = 1	d = 2	d=3
10	0.0002146469778381288	0.00011674399138428271	N/A	N/A
25	0.00014216199633665383	0.00019826999050565064	0.000396706978790462	N/A
50	0.00016400401364080608	0.00021228697733022273	0.0011697070149239153	N/A
100	0.0001228849869221449	0.00015309100854210556	0.00028948902036063373	0.001067980017978698

FullLeaves:

N elements	Running Time
10	2.4834007490426302e-05
25	3.0291994335129857e-05
50	4.3710024328902364e-05
100	5.319601041264832e-05

Conclusion

In this lab we learned to implement basic BTree operations. I learned that some methods will run linearly and others will run extremely fast whether we have 10 or 100 items in the BTree. I also learned quick ways to test code i a manner where only certain variables are manipulated in order to keep everything as relative as possible. I really enjoyed this lab due to being able to feel confident with all if not most BTree operations. Some implementations lacked since the experimental results did not yield as needed, in the future perhaps better tracing might improve experimental results accuracy in such methods.

Appendix

```
#Course: CS 2302 Data Structures | Spring 2019
#Author: Maria Fernanda Corona Ortega
#Assignment: Lab 4
#Instructor: Olac Fuentes
#Purpose of Code: The purpose of this code is to implement and modify
basic
#BTree functions with a BTree of size 5
#Last Modification: 04/05/2019 1:39pm
import random
import timeit

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

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)

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()

def IsFull(T):
    return len(T.item) >= T.max_items

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)


def height(T):
    if T.isLeaf:
        return 0
    return 1 + height(T.child[0])


def Search(T,k):
    # Returns node where k is, or None if k is not in the tree
    if k in T.item:
        return T
    if T.isLeaf:
        return None
    return Search(T.child[FindChild(T,k)],k)

def Print(T):
    # Prints items in tree in ascending order
    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)])

def PrintD(T,space):
    # Prints items and structure of B-tree
    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+'   ')

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)

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

def CompHeight(T):#Computes height of Tree
    if T.isLeaf:
        return 0
    return 1 + CompHeight(T.child[0])

def ToList(T, L):#Extracts elements into sorted list
    if T.isLeaf:
        for t in T.item:
            L.append(t)
    else:
        for i in range(len(T.item)):
            ToList(T.child[i], L)
            L.append(T.item[i])
        ToList(T.child[len(T.item)], L)
    return L

def MinAtDepth(T,d):#Returns minimun element at depth
    if d == 0:
        return T.item[0]
    else:
        return MinAtDepth(T.child[0], d-1)

def MaxAtDepth(T,d):#Returns maximum element at depth
    if d == 0:
        return T.item[len(T.item)-1]
    else:
        return MaxAtDepth(T.child[len(T.child)-1], d-1)

def NodesAtDepth(T,d): # Returns number of nodes at given depth
    count = 0
    if d == 0: #root located at height 0 will contain a single node
        return 1
    if T.isLeaf:
        return len(T.child)
    else:
        for i in T.child:
            count += NodesAtDepth(i,d-1)
    return count

def PrintAtDepth(T,d):#Prints all items in tree at given depth
    if d == 0:
        print(T.item[:])
    else:
        for i in T.child:
            PrintAtDepth(i, d-1)

def FullNodes(T):#FIXME Returns number of nodes that are full
    count = 0
    if T is None:
        return 0
    if T.isLeaf:
        if len(T.item) == T.max_items:
            return 1
        else:
            return 0
    else:
        for i in T.child:
            count += FullNodes(i)
    return count

def FullLeaves(T):#Returns number of leaves that are full
    count = 0
    if T.isLeaf:
        if len(T.item) == T.max_items:
            return 1
        else:
            return 0
    else:
        for i in T.child:
            count += FullLeaves(i)
    return count

def FindDepth(T,k):#FIXME Given a key returns depth at which it is
found or -1 if not there
    depth = 0
    if T is None:
        return -1
    if T.isLeaf: #I item is not found returns -11, needs to return -1
        if k in T.item:
            return 1
        else:
            return 0
    else:
        depth =+ FindDepth(T.child[FindChild(T,k)],k)
    return depth

##########################TESTIING AND
IMPLEMENTATION########################
L = []
T = BTree()

size = 100

for i in range(size):
    L.append(random.randint(0, size*2))

for i in L:
    print('Inserting',i)
    Insert(T,i)
    PrintD(T,'')
    #Print(T)
    print('\n####################################')
```

```
##########################TESTIING
COMPHEIGHT##################################

start = timeit.default_timer()

print(CompHeight(T))

stop = timeit.default_timer()

print('CompHeight Execution: ', stop - start)

print()

###############################TESTIING
TOLIST###################################

NL =[]

start = timeit.default_timer()

ToList(T, NL)

stop = timeit.default_timer()

print('ToList Execution: ', stop - start)
print()

###############################TESTIING
MINATDEPTH###################################

for d in range(CompHeight(T)+1):
    start = timeit.default_timer()

    print("MIN at depth",d,": ",MinAtDepth(T,d))

    stop = timeit.default_timer()

    print('MinAtDepth Execution: ', stop - start)
    print()

###############################TESTIING
MAXATDEPTH###################################

for d in range(CompHeight(T)+1):
    start = timeit.default_timer()
    print("MAX at depth",d,": ",MaxAtDepth(T,d))

    stop = timeit.default_timer()

    print('MaxAtDepth Execution: ', stop - start)
    print()

###############################TESTIING
NODESATDEPTH###################################

for d in range(CompHeight(T)+1):
    start = timeit.default_timer()

    print("Nodes at depth",d,": ",NodesAtDepth(T,d))

    stop = timeit.default_timer()

    print('NodesAtDepth Execution: ', stop - start)
    print()

###############################TESTIING
PRINTATDEPTH###################################

for d in range(CompHeight(T)+1):
    start = timeit.default_timer()

    print("Items at depth",d,": ")
    PrintAtDepth(T,d)

    stop = timeit.default_timer()

    print('PrintAtDepth Execution: ', stop - start)
    print()


##########################TESTIING
FULLLEAVES###################################

start = timeit.default_timer()

print(FullLeaves(T))

stop = timeit.default_timer()

print('FullLeaves Execution: ', stop - start)
print()
```

Academic Honesty Certification
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.

Maria F. Corona Ortega

[illegible]

09/09/2017