**Course Name:** 2302 **Author:** Olugbenga Iyiola **ID:** 80638542 **Instructor:** Olac Fuentes **TA:** Nath Anindita/ Malileh Zargaran **LAB #4 Report**

**Introduction**

The purpose of this lab is to implement basic B- tree operations, including finding height, maximum and minimum element, number of nodes and printing all elements at a given depth. It also includes extraction of elements of a b-tree into a sorted and finding the total number of full nodes and leaves it contains.

A b-tree is an organizational structure for information storage and retrieval in the form of a tree in which all terminal nodes are the same distance from the base, and all nonterminal nodes have between *n* and 2 *n* subtrees or pointers (where *n* is an integer).

In [computer science](https://en.wikipedia.org/wiki/Computer_science), a B-tree is a self-balancing [tree data structure](https://en.wikipedia.org/wiki/Tree_data_structure) that maintains sorted data and allows searches, sequential access, insertions, and deletions in [logarithmic time](https://en.wikipedia.org/wiki/Logarithmic_time). The B-tree is a generalization of a [binary search tree](https://en.wikipedia.org/wiki/Binary_search_tree) in that a node can have more than two children. Unlike [self-balancing binary search trees](https://en.wikipedia.org/wiki/Self-balancing_binary_search_tree), the B-tree is well suited for storage systems that read and write relatively large blocks of data, such as discs. It is commonly used in [databases](https://en.wikipedia.org/wiki/Database) and [file systems](https://en.wikipedia.org/wiki/File_system). (Wikipedia)

**Proposed Solution Design and Implementation**

The b-tree is implemented with python native list with the various operations carried out in O(log n) time as expected and the pseudocodes for the various implementations are given below;

**Find height of B-tree**

if node is Leaf:

return 0

return 1 + recursive call on 1st node’s child # Increment by one for every next child

**Extracting BTree to SortedList**

if node is Leaf:

for i in range length of node’s item

Append leaf node items into list recursively

else:

for i in range length of node’s item

Recursive call on children to append their items

Appending current node items

Then Recursive call to append items of node's last child

return Sorted List

**Finding Minimum At Given Depth**

Checking if depth is 0 which will be the needed depth

return node’s first item

Check if node is Leaf:

return infinity if depth is not in tree or valid

else:

return recursive call using pointer to first child

**Finding Maximum At Given Depth**

Checking if depth is 0 which will be the needed depth

return node’s last item

Check if node .is Leaf:

return infinity if depth is not in tree or valid

else:

return Recursive call using pointer to last child

**Finding number of nodes at a given depth**

Create counter

Check if depth is 0 which will be at the needed depth

return 1

else

for i in range length of node’s child

count number of node at given depth

return counter

P**rint all items in tree at a given depth**

Checking if depth is 0 which will be at the needed depth

for i in range of length node’s items

Printing node items at given depth

else:

for i in range of length node’s items

Recursive on node's children

Recursive on node's last child

**Finding number of nodes in the tree that are full**

Create Counter to count full nodes

Check if node is not Leaf:

for every child of node:

count by making recursive call on every child

if length of node’s item == maximum items of node

Increment counter by 1 for every full node

Return counter

**Finding number of Full Leaves**

Create counter

Check if leaf is full

return 1 if full

else:

for i in range of length of node’s children:

count by making recursive call on node’s children

return counter

**Finding the depth at which a given key, k is found**

Checking if key is in node

return 0

if node is leaf

return -1 to indicate k is not in tree

Check if k is greater that last item of node

If true make recursive call on last child of node

else:

for i in range length of node’s items

Check for correct index of node's child where k can be found

Find depth by making recursive call on node's children

Break out of loop if k is found

if d == -1:

return -1 indicating k is not in tree

return depth + 1

**Experimental Result**

System Specification: HP Windows 10, 1.60GHZ Intel® Celeron® , 4.GB RAM, 64-bit operating system

The results of the various test cases for each of the algorithms are shown below:

**Find height of B-tree**

|  |  |
| --- | --- |
| **N(Input)** | **Runtime in nanoseconds** |
| **20** | **74243** |
| **100** | **72962** |
| **200** | **74242** |
| **500** | **79363** |
| **1000** | **78083** |

Recurrence equation is T(N)= T(N-1)+ 1 and Running time is O(log N)

**Extracting BTree to SortedList**

|  |  |
| --- | --- |
| **N(Input)** | **Runtime in nanoseconds** |
| **20** | **37761** |
| **100** | **114564** |
| **200** | **264328** |
| **500** | **1120036** |
| **1000** | **3641717** |

Recurrence Equation: T(N) = 2NT(N-1) + N which, using the master theorem, gives us O(log N).

**Finding Minimum At Given Depth**

|  |  |
| --- | --- |
| **N(Input)** | **Runtime in Nanoseconds** |
| **20** | **72322** |
| **100** | **72963** |
| **200** | **73602** |
| **500** | **79362** |
| **1000** | **73602** |

Recurrence Equation: T(N) = T(N-1)+ 1 which, using the master theorem, gives us O(log N).

**Finding Maximum At Given Depth**

|  |  |
| --- | --- |
| **N(Input)** | **Runtime in Nanoseconds** |
| **20** | **72322** |
| **100** | **72322** |
| **200** | **74882** |
| **500** | **76803** |
| **1000** | **72962** |

Recurrence Equation: T(N) = T(N-1)+ 1 which, using the master theorem, gives us O(log N).

**Number of Nodes at Given Depth in a B-tree**

|  |  |
| --- | --- |
| **N(Input)** | **Runtime in Nanoseconds** |
| **20** | **85122** |
| **100** | **109444** |
| **200** | **95363** |
| **500** | **104323** |
| **1000** | **87043** |

Recurrence Equation: T(N) = NT(N-1)+ 1 which, using the master theorem, gives us O(log N).

**Printing all Items at a Given Depth**

|  |  |
| --- | --- |
| **N(Input)** | **Runtime in Nanoseconds** |
| **20** | **548498** |
| **100** | **2864732** |
| **200** | **1458607** |
| **500** | **1323563** |
| **1000** | **704662** |

Recurrence Equation: T(n) = NT(N-1)+ 1 which, using the master theorem, gives us O(n).

**Finding Number of Full Nodes in a B-tree**

|  |  |
| --- | --- |
| **N(Input)** | **Runtime in Nanoseconds** |
| **20** | **82563** |
| **100** | **107523** |
| **200** | **161285** |
| **500** | **344971** |
| **1000** | **630420** |

Recurrence Equation: T(n) = NT(N-1)+ N which, using the master theorem, gives us O(n).

**Finding Number of Full Leaf Nodes in a B-tree**

|  |  |
| --- | --- |
| **N(Input)** | **Runtime in Nanoseconds** |
| **20** | **95363** |
| **100** | **156165** |
| **200** | **241288** |
| **500** | **636180** |
| **1000** | **1353004** |

Recurrence Equation: T(n) = NT(N-1)+ N which, using the master theorem, gives us O(n).

**Finding the depth at which a given key, k is found**

|  |  |
| --- | --- |
| **N(Input)** | **Runtime in Nanoseconds** |
| **20** | **78722** |
| **100** | **76162** |
| **200** | **78722** |
| **500** | **78722** |
| **1000** | **79363** |

Recurrence Equation: T(N) = NT(N-1)+ N which, using the master theorem, gives us O(n).

**CONCLUSION**

B-trees are not very efficient, when compared to other balanced trees, when they reside in RAM. Inserting or deleting elements involve moving many keys/values around (Quora), however B-Tree provides ordered sequential access to the index. You can iterate over the items in a B-Tree much like binary trees provides. Iteration over a B-Tree provides the items or keys in ascending (or descending) order.

**Appendix**

***Programmed by Olac Fuentes***

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)

L = [30, 50, 10, 20, 60, 70, 100, 40, 90, 80, 110, 120, 1, 11 , 3, 4, 5,105, 115, 200, 2, 45, 6]

T = BTree()

for i in L:

print('Inserting',i)

Insert(T,i)

PrintD(T,'')

#Print(T)

print('\n####################################')

SearchAndPrint(T,60)

SearchAndPrint(T,200)

SearchAndPrint(T,25)

SearchAndPrint(T,20)

print(height(T))

**Wikipedia**

[**https://en.wikipedia.org/wiki/Sorting\_algorithm#Comparison\_of\_algorithms**](https://en.wikipedia.org/wiki/Sorting_algorithm#Comparison_of_algorithms)

**Academic Dishonesty**

This work was done by me without any act or practice of academic dishonesty

**SIGNATURE**

**OLUGBENGA IYIOLA(OT)**

**……………………………………………….**