David Amparan

CS 2302

Instructor: Professor -Fuentes, Olac

T.A : Anindita Nath, Maliheh Zargaran

March 20, 2019

**Introduction**

B-Trees are an ADT much like binary trees when storing data and attaining it as well with a few twists. Instead of storing only one item within the nodes we can store a list of items within each node and due to this slight change, we always attain a balanced tree. For our lab assignment we tackle on creating methods to see the behavior and functionality benefits of this ADT.

**Solution**

compHeight – To compute the height of the B Tree we must traverse the nodes until we hit the leaves and count for each traversal and return this number

intoSortList – To attain a sort list we travel the tree in order, and add each array onto one another (left, root,right) up until we reach the rightmost leaf

minDepth – To attain the minum on a depth we traverse to that depth by comparing that depth key to zero (meaning we are there) then once we are there, we return the first item in that list since the lists are in order

maxDepth – Very similar to the minDepth, we compare a depth key to zero, and if that depth does exist, we return the last item within that node list

nodesAtDepth – Compare a depth key to zero, if we reach that condition then we count how many nodes reached that point, thus giving us the nodes at that depth

printDepth – Once again using the depth key and comparing it to zero, if this case is then reached, we print the list of these nodes

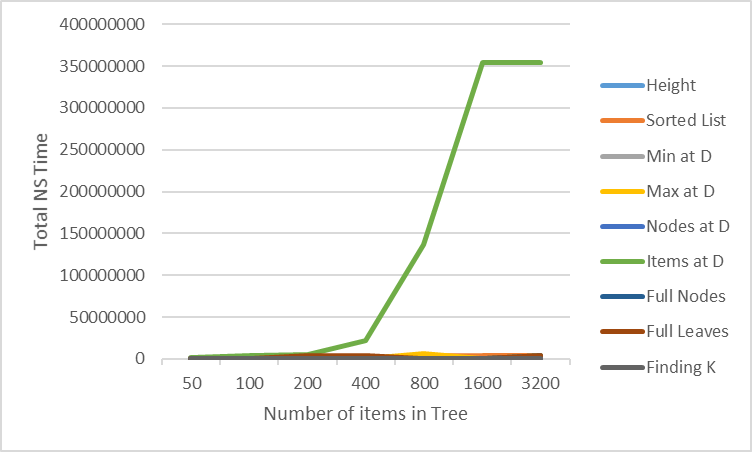
fullNodes – We traverse the entire B Tree and within each node we visit we check if the item is full, then we return 1 if it is else 0

fullLeaves – We traverse the tree until we reach the leaves, we return 1 for those that are full

depthOfK – We traverse the entire tree and look for k in list if it is found then we return the depth of that. This is done by keeping a key of the depth and increasing with each recursive call

**Experimental**

To fully test our methods, we must make use of the utility of the tree structure. Which means, we must traverse the tree in 3 different ways, see the entire tree, see a portion, or a certain depth. Our methods are designed around this logic and thus we must increase the size of the tree by *n* elements as well as seeing/searching (if applicable) in the deepest part of the tree which would be its leaves. Therefor the test cases I chose are going from a size of 50-3200 elements in the tree and for those with a depth variable, the depth is the max.



As we can see the only true item that grew rapidly was that of printing items at a certain depth, all others recorded a minimal growth rate due to the log(n) time complexity of the B-Tree thanks to its structure. The structure allows us to only traverse one branch if desired as well as possibly finding or attaining the needed data much faster than with a list.

**Conclusion**

With this lab assignment the benefits of the B-Tree as well as the drawbacks were exposed by creating utility methods for the handling the data. With many of the methods run time being O(log n) it is a very useful and efficient way of storing and handling the data. The downside of the B Tree is the levels of complexity one must increase the more complex a problem gets. From only needing to ignore one side of the tree to having to traverse the entire tree all the way to its leaf nodes. To wrap it all into one, a B-Tree is a more effective binary tree that depending on the functionality one wants to use it can either be effective or ineffective, like many of the ADT’s.

**Appendix**

I, David Amparan, assure that the source code was done by me and only me. With no assistance from the internet nor other classmates. If anything is found that can be labeled as academic dishonesty, I will assume full responsibility for it.

David Amparan

Source Code:

|  |
| --- |
| 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 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) |
|  |  |
|  | """ |
|  | Method Name: compHeight Parameters: T |
|  | Functinality: It will compute the height of the tree and return that number |
|  | """ |
|  | def compHeight(T): |
|  | if T.isLeaf: |
|  | return 0 |
|  | return compHeight(T.child[0])+1 |
|  |  |
|  |  |
|  | """ |
|  | Method Name: intoSortlist | Parameters: B Tree |
|  | Funtionalty: For this method we will traverse the B-Tree and |
|  | extract the items in a sorted manner and store them within a list |
|  | """ |
|  |  |
|  | def intoSortList(T): |
|  | if T.isLeaf: |
|  | return T.item |
|  | #after returning the keafs we must concatanate the root to the other parts |
|  | ourList = [] |
|  | givenBack = [] |
|  | #here we append the left side then root then the next and so on so forth |
|  | for i in range(len(T.item)): |
|  | givenBack = intoSortList(T.child[i]) |
|  | ourList = ourList + givenBack |
|  | ourList = ourList + [T.item[i]] |
|  | #this recursive call is to traverse the right side of our B Tree |
|  | ourList = ourList + intoSortList(T.child[len(T.item)]) |
|  | return ourList |
|  |  |
|  | """ |
|  | Method Name: minDepth | Parameters: BTree and Depth |
|  | Functionality: Given a BTree find the mininum element at that given depth |
|  | """ |
|  | def minDepth(T, d): |
|  | if d==0: |
|  | return T.item[0] |
|  | #if d is not zero and we are at a leaf then the depth exceeds the height of the tree |
|  | if T.isLeaf: |
|  | return -1 |
|  | return minDepth(T.child[0], d-1) |
|  |  |
|  |  |
|  | """ |
|  | Method Name: maxDepth | Parameters: B Tree, Depth |
|  | Parameters: Similar to the minDepth method we will find the max element |
|  | at a given depth |
|  | """ |
|  | def maxDepth(T,d): |
|  | if d==0: |
|  | return T.item[-1] |
|  | if T.isLeaf: |
|  | return -1 |
|  | return maxDepth(T.child[-1],d-1) |
|  |  |
|  |  |
|  | """ |
|  | Method Name: nodesAtDepth(T,d) | Parameters: BTree and Depth |
|  | Functionality: We will return the number of nodes at that given depth and if the |
|  | depth is not found then we will return -1 |
|  | """ |
|  | def nodesAtDepth(T,d): |
|  | if d==0: |
|  | return 1 |
|  | if T.isLeaf: |
|  | return 0 |
|  | nodes = 0 |
|  | for i in range(len(T.item)): |
|  | nodes = nodes + nodesAtDepth(T.child[i], d-1) |
|  | nodes = nodes + nodesAtDepth(T.child[len(T.item)],d-1) |
|  | return nodes |
|  |  |
|  |  |
|  | """ |
|  | Method Name: printDepth | Parameters: T, D |
|  | Functionality: The method will traverse through a given depth |
|  | and if it exists, it will print all items at that depth |
|  | """ |
|  | def printDepth(T, d): |
|  | if d==0: |
|  | for i in T.item: |
|  | print(i,',',end=' ') |
|  | if T.isLeaf: |
|  | return |
|  | #to traverse both sides of the tree require a forloop |
|  | for i in range(len(T.item)): |
|  | printDepth(T.child[i],d-1) |
|  | printDepth(T.child[len(T.item)], d-1) |
|  |  |
|  |  |
|  | """ |
|  | Method Name: fullNodes | Parameters: T |
|  | Functionality: fullNodes will traverse the entire tree and |
|  | determine which nodes are full, if the nodes are full then we will return 1 |
|  | Will count all full nodes in the tree |
|  | """ |
|  | def fullNodes(T): |
|  | #initial comparison |
|  | if T.isLeaf: |
|  | if len(T.item)==5: |
|  | return 1 |
|  | else: |
|  | return 0 |
|  | if len(T.item)==5: |
|  | return 1 |
|  |  |
|  | #recursive calls to traverse left side and right side |
|  | total = 0 |
|  | for i in range(len(T.item)): |
|  | total = total + fullNodes(T.child[i]) |
|  | total = total + fullNodes(T.child[len(T.item)]) |
|  | return total |
|  |  |
|  | """ |
|  | Method Name: fullLeaves | Parameters: B Tree |
|  | FUnctionality: Will count the amount of full leaves and return that number |
|  | """ |
|  | def fullLeaves(T): |
|  | #we only care about the leaves |
|  | if T.isLeaf: |
|  | if len(T.item)==5: |
|  | return 1 |
|  | else: |
|  | return 0 |
|  | total = 0 |
|  | for i in range(len(T.item)): |
|  | total = total + fullLeaves(T.child[i]) |
|  | total = total + fullLeaves(T.child[len(T.item)]) |
|  | return total |
|  |  |
|  | """ |
|  | Method Name: depthOfK | Parameters: T, k, d |
|  | Functionality: The method will run through the entire tree while keeping |
|  | tabs on the depth traveled to return that depth if the item is found |
|  | """ |
|  | def depthOfK(T,k,d): |
|  | #if we end up at a leaf we check if it is within |
|  | if T.isLeaf: |
|  | if k in T.item: |
|  | return d |
|  | else: |
|  | return -1 |
|  | #now we check incase it is within a seperate node |
|  | if k in T.item: |
|  | return d |
|  |  |
|  | #recursive call |
|  | return depthOfK(T.child[FindChild(T,k)], k, d+1) |
|  |  |
|  |  |
|  |  |
|  |  |
|  | L = [30, 50, 10, 20, 60, 70, 100, 40, 90, 80, 110, 120, 1, 11 , 3, 4, 5,105, 115, 200, 2, 45, 6] |
|  | #L = [30, 50, 10, 20, 80, 110, 120, 1, 11 , 3, 4, 5,105, 115, 200, 2, 45, 6] |
|  | T = BTree() |
|  |  |
|  | for i in L: |
|  | Insert(T,i) |
|  |  |
|  |  |
|  | print("Our B-Tree") |
|  | PrintD(T, '') |
|  | print("--------------------------------------") |
|  | print("Computing Height of the Tree") |
|  | print("Height:", compHeight(T)) |
|  | print() |
|  | print("--------------------------------------") |
|  | print() |
|  | print("Sorted List from the Tree") |
|  | print("List:", intoSortList(T)) |
|  | print() |
|  | print("--------------------------------------") |
|  | print() |
|  | d = 1 |
|  | print("Minimum value at depth", d) |
|  | print("Minimum at depth", d,":", minDepth(T,d)) |
|  | print() |
|  | print("--------------------------------------") |
|  | print() |
|  | print("Maximum value at depth", d) |
|  | print("Maximum at depth", d,":", maxDepth(T,d)) |
|  | print() |
|  | print("--------------------------------------") |
|  | print() |
|  | print("Total number of Nodes at a Depth") |
|  | print("Nodes at depth", d, ":", nodesAtDepth(T,d)) |
|  | print() |
|  | print("--------------------------------------") |
|  | print() |
|  | print("Printing the elements of certain depth") |
|  | printDepth(T,d) |
|  | print() |
|  | print("--------------------------------------") |
|  | print() |
|  | print("The number full Nodes") |
|  | print("Full Nodes:", fullNodes(T)) |
|  | print() |
|  | print("--------------------------------------") |
|  | print() |
|  | print("Number of full Leaves") |
|  | print("Full Leaves", fullLeaves(T)) |
|  | print() |
|  | print("--------------------------------------") |
|  | print() |
|  | print("The Depth of an item present in the B-Tree") |
|  | k = 50 |
|  | d = 0 |
|  | print("Searching for", k, ", depth:", depthOfK(T,k,d)) |