Jon Munoz

Lab 2

CS2302

Professor: Olac Fuentes

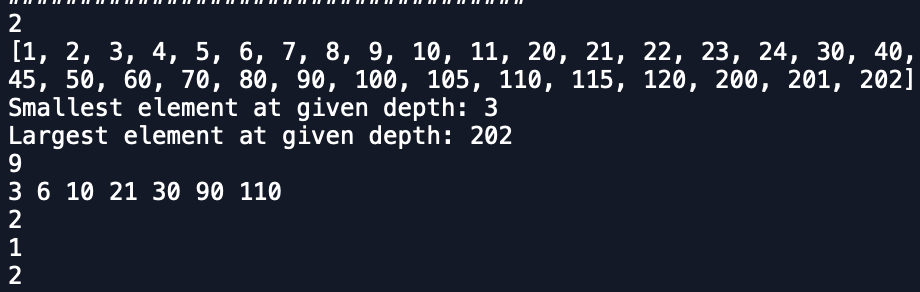
**INTRODUCTION**

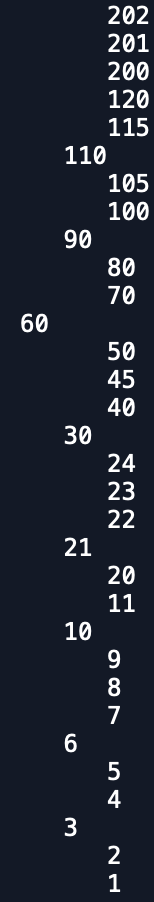
In rhis lab we were tasked with working with B-Trees. There were nine different questions that we had to work on: computing height, converting a B-Tree to an ordered list, finding the minimum element at a given depth, finding the largest element at a given depth, returning all nodes at a given depth, printing all the items at a given depth, returning the number of full nodes in the tree, returning the number of full leaves in the tree, and returning the depth of a given key.

**PROPOSED SOLUTION DESIGN AND IMPLEMENTATION**

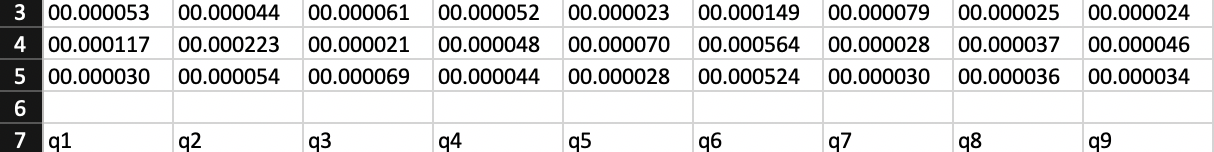
These questions were quite simple due to the fact that I had basically completed them in the take home quiz we were assigned. For the height of the tree since all leaf nodes are at the same depth I simply had to traverse down any given path until I was at a leaf node. For the conversion to list I had a bit of trouble. What I ended up doing was creating four list, and empty one, one containing child nodes, one containing items in each child, and a list concatenated with all of them, and concatenating them at the end of the method and returning that concatenated list. For the smallest at a given depth what I did was recursively called my method until my depth variable was zero and once I got there I returned the far left value. For the largest at a given depth I did what I did for the smallest but instead of returning the far left value I returned the far right value. For number of nodes at a given depth what I did was make traverse down until my depth parameter was 0 and I would return one. I also had a for loop to go down each child so I would get a return value of one for each node at the depth. For the printing at a given depth I did something very similar to the number of nodes at a depth however instead of returning anything I would print the items in each node. For the full nodes method I traversed the entire tree and compared the length of each node to max\_items and if the length was equal to the max\_items value then I would return 1 else I would do nothing and move on. Full leaf nodes was very similar to the previous part except I only did comparisons when I was at a leaf node. For the find depth of k what I did was compare k to the first item in a given node and traverse down the tree based on if k was less than of greater to the item while also incrementing a counter variable so I can return that value when and if I find K.

**EXPERIMENTAL RESULTS**

Below are the results of my method calls in order from method one to method nine. I have also included a picture of the tree that I used for all of my method calls in order to visually see if the methods returned what they were supposed to. For the height of this tree I got the value two which is the correct value that should have been returned since the height of the tree is two. For the tree to list I printed the list which is in order which is the correct was that the list should have come out. For the smallest element at a given depth I ran the method with depth one so therefore three should have been returned which it was. For the largest element at a given depth I ran the method with the depth two and 202 was returned which is the correct output. For the number of nodes at a given depth I ran the method with depth two and the method returned nine which is the correct output. For printing at a depth I ran the method with one for my depth and the correct numbers were printed. For full nodes that correct output of two was returned since there where two full nodes in this tree that I used. For full leaf nodes one was returned which is the correct output. For finding the depth of a value I ran the method with 70 which was in depth two and my method correctly returns that value.



Below are the run times of the different methods with the y axis being the number of nodes in the tree and the x axis being each question.

**CONCLUSION**

At the end of this assignment I got more familiar with B-Trees and how they work. I am able to work with them more comfortable even though some little issues may arise here and there

**APPENDIX**

**SOURCE CODE**

|  |
| --- |
| #Jon Munoz |
|  | #CS2302 Data Structures |
|  | #Lab 4 |
|  | #Instructor:Olac Fuentes |
|  | #TA:Anindita Nath, Maliheh Zargaran |
|  | #Last Modified 3/15/19 |
|  |  |
|  | # Code to implement a B-tree |
|  | # Programmed by Olac Fuentes |
|  | # Last modified February 28, 2019 |
|  |  |
|  | 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) |
|  |  |
|  | ##################################################### |
|  |  |
|  | #this method will find the height of a given B-Tree |
|  | def height(T): |
|  | if T.isLeaf:#if the current node is a you want to return 0 |
|  | return 0 |
|  | return 1 + height(T.child[0])#if the node is not a leaf then you want to add 1 to the height value |
|  |  |
|  | #Dthis method converts a B-Tree to a sorted list |
|  | def TtoL(T): |
|  | if T.isLeaf:#when the node is a leaf you want to return a list |
|  | List = list(T.item)#the returned list is a list of the current node |
|  | return List |
|  | else: |
|  | L = []#create an empty list to add to |
|  | for i in range(len(T.item)):#for loop will populate two differant arrays L1 and L2 |
|  | L1 = TtoL(T.child[i])#L1 is filled with each child node |
|  | L2 = [T.item[i]]#L2 is populated with each item node |
|  | L = L + L1 + L2#L is then made to be the concatination of L,L1, and L2 |
|  | L4 = TtoL(T.child[-1])#L4 must be called with the last child node since the for loop does not go to it |
|  | return L + L4#add L4 to the rest of the created list |
|  |  |
|  |  |
|  | #this method goes to a given depth and returns the smallest number at that depth |
|  | def SmallestAtDepth(T, d): |
|  | if T == None:#if T is none then the item was not found and return none |
|  | return None |
|  | if d == 0:#if d is 0 then you are at your desired depth |
|  | return T.item[0]#return the furthest left element |
|  | return SmallestAtDepth(T.child[0], d - 1)#call the method recursively with the leftmost child |
|  |  |
|  | #this method goes to a given depth and returns the largest number at that depth |
|  | def LargestAtDepth(T, d): |
|  | if T == None:#if T is none then the item was not found and return none |
|  | return None |
|  | if d == 0:#if d is 0 then you are at your desired depth |
|  | return T.item[len(T.item) - 1]#return the furthest right element |
|  | return LargestAtDepth(T.child[len(T.item)], d - 1)#call the method recursively with the rightmost child |
|  |  |
|  |  |
|  | #this method returns the number of nodes at a given depth |
|  | def NodesAtDepth(T, d): |
|  | if T is None:#if T is none then there is no node and you return 0 |
|  | return 0 |
|  | if d == 0:#if d is 0 then you are at the desired depth so you return 1 |
|  | return 1 |
|  | else: |
|  | num = 0#num is the total number of nodes (which is initially 0) |
|  | for i in range(len(T.child)):#for loop goes to all children |
|  | if T.child != None: |
|  | num += NodesAtDepth(T.child[i], d - 1)#if the child is not none then you want to call the method recursively with each child |
|  | return num#return the total |
|  |  |
|  | #this method prints all the items at a given depth |
|  | def PrintAtDepthD(T, d): |
|  | if d > height(T):#if the given depth is larger than the height of the tree then there are no nodes |
|  | print("No nodes at this depth")#print that there are no nodes |
|  | return |
|  | if d == 0:#if d is 0 then you are at the desired depth so you print all the items |
|  | for i in range(len(T.item)):#for loop prints all items |
|  | print(T.item[i], end = ' ') |
|  | else: |
|  | for i in range(len(T.child)):#for loop goes through every child |
|  | PrintAtDepthD(T.child[i], d - 1) |
|  |  |
|  | #this method gets the total number of nodes in the tree that are full and returns the value |
|  | def FullNodes(T): |
|  | if len(T.item) == T.max\_items:#if the length of a given node is equal the max\_items value then it is full so return 1 |
|  | return 1 |
|  | else: |
|  | total = 0#total keeps track of the total number of full nodes |
|  | for i in range(len(T.child)):#for loop goes through every child node |
|  | total = total + FullNodes(T.child[i])#add to total |
|  | return total |
|  |  |
|  | #this method will find the total number of leaf nodes that are full |
|  | def FullLeafNodes(T): |
|  | if T.isLeaf and len(T.item) == T.max\_items:#if a node is a leaf and is full then you want to return 1 |
|  | return 1 |
|  | else: |
|  | total = 0#total keeps track of the total number of full leaf nodes |
|  | for i in range(len(T.child)):#for loop goes through every child node |
|  | total = total + FullLeafNodes(T.child[i])#add to total |
|  | return total |
|  |  |
|  | #this method will find a given number if it is in the tree and return its depth |
|  | def FindDepth(T, k): |
|  | if k in T.item:#if the key is in the current node then you return 0 |
|  | return 0 |
|  | depth = 0#depth initially initialized to 0 |
|  | if k > T.item[len(T.item)-1]:#if the key is greater than the last item go to last child |
|  | depth = FindDepth(T.child[len(T.item)], k) |
|  | elif k < T.item[0]:#if key is less than the first item travel to the leftmost child |
|  | depth = FindDepth(T.child[0], k) |
|  | if depth >= 0:#if depth is greater than or equal to 0 add 1 to depth |
|  | return 1 + depth |
|  | else: |
|  | return -1#if the key is not found return -1 |
|  |  |
|  |  |
|  | L = [30, 50, 10, 20, 60, 70, 100, 40, 90, 80, 110, 120, 1, 11 , 3, 4, 5,105, 115, 200, 2, 45, 6,7,8,9,201,202,21,24,22,23] |
|  | T = BTree() |
|  | for i in L: |
|  | #print('Inserting',i) |
|  | Insert(T,i) |
|  | #PrintD(T,'') |
|  | #Print(T) |
|  | #print('\n####################################') |
|  |  |
|  | PrintD(T, '') |
|  | print('\n####################################') |
|  |  |
|  | #Below are the method calls |
|  | print(height(T)) |
|  | print(TtoL(T)) |
|  | print("Smallest element at given depth:", SmallestAtDepth(T, 1)) |
|  | print("Largest element at given depth:", LargestAtDepth(T, 1)) |
|  | print(NodesAtDepth(T,2)) |
|  | PrintAtDepthD(T, 1) |
|  | print() |
|  | print(FullNodes(T)) |
|  | print(FullLeafNodes(T)) |
|  | print(FindDepth(T, 70)) |

“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 provide inappropriate assistance to any student in the class.”

