**Author:** Julian Gonzalez

**Assignment:** Lab 4 Report

**Course:** CS 2302 - Data Structures 10:30-11:50

**Instructor:** Fuentes, Olac

**T.A.:** Nath, Anindita

***Introduction***

The Purpose of this lab is to further expand the capabilities of given B-tree code. We are practicing finding the depths of certain elements in the B-Tree

***Solutions***

For finding the height of the B-tree I wrote a recursive program. The program had the base case of if T (being the name of the B-tree) is leaf it will return 0 otherwise the program will do a recursive call of 1+…. +0 while iterating through the children of the tree

For the extractor function I had the base case of the tree being a leaf it would return L.extend(T.item) (extend instead of append), as this would add the items of the current node into the list that we are creating, next was a for loop that would iterate through the children of the list and would call the extractor function recursively.

Largest at depth I had two base cases. The first would check the parameter d and if it equals to 0 and if it was it would return the current item at -1 in the list which should translate into the largest item. The second base case would check if the current node is a leaf and if it was it would return -maf.inf as the depth we are looking for does not exist. Finally, the last case would do a recursive call that would iterate to the -1 place of the child and decrease d by 1

Smallest at depth I had two base cases. The first would check the parameter d and if it equals to 0 and if it was it would return the current item at 0 in the list which should translate into the smallest item. The second base case would check if the current node is a leaf and if it was it would return -maf.inf as the depth we are looking for does not exist. Finally, the last case would do a recursive call that would iterate to the 0 place of the child and decrease d by 1

Number at depth I had two base cases. The first would check the parameter d and if it equals to 0 and if it was it would create a count c=0 which would then be inside a for loop that would count the number of items in the current node and then return c. The second base case would check if the current node is a leaf and if it was it would return -maf.inf as the depth we are looking for does not exist. Then finally the last case would check again if t is not a leaf and would do a counter c=0 and iterate through the children of the tree and in a for loop and decrease d by 1 every time and in the call, c would be added with the returned value and then we would return c at the end.

Print at a certain depth which had to cases. The first would check the parameter d and if it equals to 0 and if it was it go into a for loop that prints put the items of the current depth. The second case would check if t is not a leaf then it would go into a for loop that iterates through the entire tree and decrease d by 1.

Full Nodes and full leaf’s are very similar. For both of them I created a counter c= 0 both would check if the length of the current node is equal to max items that the B-tree can hold at each node and add one to c, the only difference is that full leaf’s would also check if the current node was a leaf or not. The second case would also check if the current node was a leaf and if it was it would return c. finally the last thing that would happen is a for loop that iterates through the entire tree and would have c+= to the recursive call.

***Experimental***

For most of my runs of the programs, I used List L = [30, 50, 10, 20, 60, 70, 100, 40, 90, 80, 110, 120, 1, 11, 3, 4, 5,105, 115, 200, 2, 45, 6]. For questions 3,4 and 5, I used the depth of 1 to find the smallest largest and printing at a certain depth for testing, question 9 I used the value of 105 to be found. I also tested all the questions with a different sized list of to see if all the functions worked correctly.

|  |  |  |  |
| --- | --- | --- | --- |
| Function Name |  | | |
|  | Trial 1 | Trial 2 | Trial 3 |
| height | ﻿0.000047 | 0.000124 | ﻿0.000074 |
| Extractor | ﻿0.000057 | ﻿0.000034 | ﻿0.000058 |
| SmallestAtDepth | ﻿0.000061 | ﻿0.000042 | ﻿0.000045 |
| LargestAtDepth | ﻿0.000056 | ﻿0.000044 | ﻿0.000041 |
| NumAtDepth | ﻿0.000020 | ﻿0.000067 | ﻿0.000023 |
| PrintAtDepth | ﻿0.000102 | ﻿0.000067 | ﻿0.000024 |
| FullNodes | ﻿﻿0.000060 | ﻿0.000029 | ﻿0.000023 |
| FullLeafs | ﻿﻿0.000023 | ﻿0.000034 | 0.000030 |
| FindDepth | ﻿0.000026 | ﻿0.000043 | ﻿0.000034 |

A screenshot of a social media post

Description automatically generatedA picture containing sky

Description automatically generated

***Conclusion***

This lab was a great way to practice more with B-Trees as they are very different when compared to Binary search trees, the given class code and functions allowed me to understand B-Trees and how they work also the functions that I created help with this.

Appendix:

1. # -\*- coding: utf-8 -\*-
2. """
3. Created on Wed Mar 13 14:22:29 2019
4. @author: Julian
5. Assignment: Lab 4
6. Instructor: Fuentes, Olac
7. T.A: Nath, Anidita
8. Purpose is to further expand the capablities of given B-tree code
9. """
10. # Code to implement a B-tree
11. # Programmed by Olac Fuentes
12. # Last modified February 28, 2019
13. **import** math
15. **class** BTree(object):
16. # Constructor
17. **def** \_\_init\_\_(self,item=[],child=[],isLeaf=True,max\_items=5):
18. self.item = item
19. self.child = child
20. self.isLeaf = isLeaf
21. **if** max\_items <3: #max\_items must be odd and greater or equal to 3
22. max\_items = 3
23. **if** max\_items%2 == 0: #max\_items must be odd and greater or equal to 3
24. max\_items +=1
25. self.max\_items = max\_items
27. **def** FindChild(T,k):
28. # Determines value of c, such that k must be in subtree T.child[c], if k is in the BTree
29. **for** i **in** range(len(T.item)):
30. **if** k < T.item[i]:
31. **return** i
32. **return** len(T.item)
34. **def** InsertInternal(T,i):
35. # T cannot be Full
36. **if** T.isLeaf:
37. InsertLeaf(T,i)
38. **else**:
39. k = FindChild(T,i)
40. **if** IsFull(T.child[k]):
41. m, l, r = Split(T.child[k])
42. T.item.insert(k,m)
43. T.child[k] = l
44. T.child.insert(k+1,r)
45. k = FindChild(T,i)
46. InsertInternal(T.child[k],i)
48. **def** Split(T):
49. #print('Splitting')
50. #PrintNode(T)
51. mid = T.max\_items//2
52. **if** T.isLeaf:
53. leftChild = BTree(T.item[:mid])
54. rightChild = BTree(T.item[mid+1:])
55. **else**:
56. leftChild = BTree(T.item[:mid],T.child[:mid+1],T.isLeaf)
57. rightChild = BTree(T.item[mid+1:],T.child[mid+1:],T.isLeaf)
58. **return** T.item[mid], leftChild,  rightChild
60. **def** InsertLeaf(T,i):
61. T.item.append(i)
62. T.item.sort()
64. **def** IsFull(T):
65. **return** len(T.item) >= T.max\_items
67. **def** Insert(T,i):
68. **if** **not** IsFull(T):
69. InsertInternal(T,i)
70. **else**:
71. m, l, r = Split(T)
72. T.item =[m]
73. T.child = [l,r]
74. T.isLeaf = False
75. k = FindChild(T,i)
76. InsertInternal(T.child[k],i)
78. **def** Search(T,k):
79. # Returns node where k is, or None if k is not in the tree
80. **if** k **in** T.item:
81. **return** T
82. **if** T.isLeaf:
83. **return** None
84. **return** Search(T.child[FindChild(T,k)],k)
86. **def** Print(T):
87. # Prints items in tree in ascending order
88. **if** T.isLeaf:
89. **for** t **in** T.item:
90. **print**(t,end=' ')
91. **else**:
92. **for** i **in** range(len(T.item)):
93. Print(T.child[i])
94. **print**(T.item[i],end=' ')
95. Print(T.child[len(T.item)])
97. **def** PrintD(T,space):
98. # Prints items and structure of B-tree
99. **if** T.isLeaf:
100. **for** i **in** range(len(T.item)-1,-1,-1):
101. **print**(space,T.item[i])
102. **else**:
103. PrintD(T.child[len(T.item)],space+'   ')
104. **for** i **in** range(len(T.item)-1,-1,-1):
105. **print**(space,T.item[i])
106. PrintD(T.child[i],space+'   ')
108. **def** SearchAndPrint(T,k):
109. node = Search(T,k)
110. **if** node **is** None:
111. **print**(k,'not found')
112. **else**:
113. **print**(k,'found',end=' ')
114. **print**('node contents:',node.item)
116. **def** height(T):##used code that was provided on class page
117. **if** T.isLeaf:
118. **return** 0
119. **return** 1 + height(T.child[0])
121. **def** Extractor(T,L):
122. **if** T.isLeaf:
123. **return** L.extend(T.item)#extend add the items if used append the list would be added
124. **for** i **in** range(len(T.child)):##for loop to iterate through th entire b tree
125. Extractor(T.child[i],L)
127. **def** LargestAtDepth(T,d):
128. **if** d==0:# base case for the depth
129. **return** T.item[-1]
130. **if** T.isLeaf:#if we reach a leaf before reaching 0 for the depth return negative infinity
131. **return** -math.inf
132. **else**:
133. **return** LargestAtDepth(T.child[-1], d-1)#searches the right most side of the tree for the largest item
135. **def** SmallestAtDepth(T,d):
136. **if** d==0 :# base case for the depth
137. **return** T.item[0]
138. **if** T.isLeaf:#if we reach a leaf before reaching 0 for the depth return negative infinity
139. **return** -math.inf
140. **else**:
141. **return** SmallestAtDepth(T.child[0],d-1)#searches the left most side of the tree for find the smallest item
143. **def** NumAtDepth(T,d):
144. **if** d ==0:# base case for the depth
145. c = 0
146. **for** t **in** T.item:#iterates the current node and counts the number of items in that depth
147. c += 1
148. **return** c
149. **if** T.isLeaf:#
150. **return** -math.inf #if we reach a leaf before reaching 0 for the depth return negative infinity
151. **if** **not** T.isLeaf:#if not leaf we iterate through the whole tree to find the current depth
152. c = 0
153. **for** i **in** range(len(T.child)):
154. c += NumAtDepth(T.child[i],d-1)
155. **return** c #returns items the the depth
156. **return** -math##i dunno what this does?
158. **def** PrintAtDepth(T,d): # Prints all items in b-tree with root T that have depth d
159. **if** d ==0:
160. **for** i **in** T.item:#prints the items of the current depth
161. **print**(i,end=' ')
162. **if** **not** T.isLeaf:#iterates the tree to find the correct depth
163. **for** i **in** range(len(T.child)):
164. PrintAtDepth(T.child[i],d-1)
165. **def** FullNodes(T):
166. c = 0
167. **if** len(T.item) == T.max\_items:#if current node is equal to the max items return add one to counter
168. c +=1
169. **if** T.isLeaf: #if the current node is a leaf return count
170. **return** c
171. **for** i **in** range(len(T.child)):#iterates the whole tree
172. c += FullNodes(T.child[i])
173. **return** c# final return for the final amount of full nodes
175. **def** FullLeafs(T):
176. c = 0
177. **if** len(T.item) == T.max\_items **and** T.isLeaf:#checks the len of current node against max items then checks if its a leaf
178. c +=1
179. **if** T.isLeaf:# if the first case is not met and is a tree return cound
180. **return** c
181. **for** i **in** range(len(T.child)):#iterate through the entire tree
182. c += FullNodes(T.child[i])
183. **return** c
185. **def** FindDepth(T,k):
186. **if** k **in** T.item:# base case if k is in the node of t
187. **return** 0
188. **if** T.isLeaf:#base case to check if t is a leaf
189. **return** -1
190. **if** k >T.item[-1]:#case to search through the right side of the Tree
191. d = FindDepth(T.child[-1],k)
192. **if** d == -1:#after searching will return -1 if d==-1
193. **return** -1
194. **return** 1+d #if not return d
195. **for** i **in** range(len(T.item)):#for loop to search rest of the tree
196. **if** k <T.item[i]:
197. d = FindDepth(T.child[i],k)
198. **if** d == 0:
199. **return** d+1
200. **if** d<0:
201. **return** -1
202. **return** d+1
204. L = [30, 50, 10, 20, 60, 70, 100, 40, 90, 80, 110, 120, 1, 11 , 3, 4, 5,105, 115, 200, 2, 45, 6]
205. #L = [1,2,3,4,5,1,1,1,1,1,6,6,6,6,6,6,2]
206. #L = [1,2,3,4,5,6,3,3,3,1,4,5,6,5,4,4,4,5,7,8,4,9,10,11,12,13,14,15]
207. T = BTree()
208. **for** i **in** L:
209. #print('Inserting',i)
210. Insert(T,i)
211. #PrintD(T,'')
212. #Print(T)
213. #print('\n####################################')
214. PrintD(T,'')
215. #1
216. **print**('\n####################################')
217. **print**(height(T))
219. #2
220. **print**('\n####################################')
221. emptyList = []
222. Extractor(T,emptyList)
223. **print**(emptyList)
225. #3
226. **print**('\n####################################')
227. **print**(SmallestAtDepth(T,1))
229. #4
230. **print**('\n####################################')
231. **print**(LargestAtDepth(T,1))
233. #5
234. **print**('\n####################################')
235. **print**(NumAtDepth(T,1))
237. #6
238. **print**('\n####################################')
239. PrintAtDepth(T,1)
241. #7
242. **print**('\n####################################')
243. **print**(FullNodes(T))
245. #8
246. **print**('\n####################################')
247. **print**(FullLeafs(T))
248. #9
249. **print**('\n####################################')
250. **print**(FindDepth(T,105))

I Julian Gonzalez 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.

* Julian Gonzalez