Gilbert Velasquez

80574186

CS2302

Lab 4 Report

Instructor: Olac Fuentes

TA’s: Anindita Nath and Maliheh Zargaran

In this lab we were to designs methods that would; compute the height, Extract the items of a B-Tree and turn it into a sorted list, get the minimum element at a given depth, get the largest element at a given depth, return the number of nodes at a given depth, print the items at a given depth, return the number of full nodes, return the number of full leaves, and return the depth of a given key node.

For the height method, since we know all leaves are at the same level we can just use the left most leaf so we just recursively add 1 until we reach a leaf.

To Extract the tree from the B-Tree we can use the same concept as extracting from a BST. All we need to do is simply apply an in order traversal. I decided to add the List as a parameter for this method so that I could make use of the .append() function. All I simply did then was traverse the list in order and that results in the sorted list. This is using the same logic as the given Print Method.

To get the Smallest at a given depth we apply the same logic as with BST. We know the smallest item is going to be the left most item or the item in index 0. So we simply get to our desired depth and return the item in the left most’s child first index. If we are given a depth that is out of bounds we return infinity. To do this we simply recur over the BST sending it the child in index 0 until we get to our desired depth, by subtracting 1 every level. If we ever reach a leaf (after checking if the item is there) we just return infinity. We can reuse this exact same logic for the largest item at a given Depth. We simply just recur with the right most child and return the value at the end of that child’s item list. We do this by using the length of the item list -1.

To get the number of nodes at a given Depth I used the logic that the number of nodes in a certain level is equal to the number of children in the level prior. So all I did was have a condition that check if the current depth -1 = 0. If that was the case I just added the length of the child list and I multiplied it by 2.

To print all the items at a given depth we simply just need to get to our depth and print the items. To do this we simply put a recur with every child until we get to our depth.

To get the number of Full Nodes in our list we first check if the node is full, if it is we return 1, next we keep track of the number by assigning the recursive call to a variable added with itself. Finally returning the variable to give us full nodes. We can recycle this logic for full Leaves by simply adding the condition after the initial check if the node is full, to see if the node is a leaf.

To get the depth of a given Key we just traverse the tree and add 1 every time we go down a level, we return -1 if we didn’t find the value in our Tree. To do this, we set a base case that returns 0 if we find the key in our current list. If we don’t find it and the list is not a leaf, we can check id the item is larger than the current max item, aka the last item in the list. If it is then we just recur over the right most list. If not, then we reoccur over the Tree. We assign a variable to the recursive call, which we will increment at the end. If d is ever -1 we just return -1 as to not mess up our math later on.

**Conclusion:**

This lab was very helpful in understanding B-Trees. This lab provided me with the knowledge to better analyze problems dealing with B-Trees. I am now able to effectively determine which of the three types of B-Tree question I am being asked which now allows me to quickly modify the template to produce the desired output. This lab was straight forward and there were no real curve balls. Unlike last time, I asked questions and through the help of the multiple in class exercises and quizzes I was able to produce a lab that I am very proud of.

**Results:**

CASE 1

Given List : [30, 50, 10, 20, 60, 70, 100, 40, 90, 80, 110, 120, 1, 11 , 3, 4, 5,200,2,4,5,201,71,25,62]

Size: 25

Height:

2

Smallest at Depth 1:

3

Largest at Depth 2:

201

Printing at Depth 2:

1 2 4 4 5 5 11 20 25 40 50 62 70 71 80 100 110 120 200 201

Number of Full Nodes:

0

Number of Full Leaves:

0

Depth of Node:

2

Number of Nodes at Depth:

6

Tree as List:

[1, 2, 3, 4, 4, 5, 5, 10, 11, 20, 25, 30, 40, 50, 60, 62, 70, 71, 80, 90, 100, 110, 120, 200, 201]

Time to Run Program: 0.026987552642822266 seconds

CASE 2:

Size: 50

Height:

2

Smallest at Depth 1:

4

Largest at Depth 2:

95

Printing at Depth 2:

1 3 5 6 8 10 13 15 17 18 25 28 32 33 35 36 40 41 44 47 49 50 53 54 63 64 70 71 75 77 80 83 90 92 95

Number of Full Nodes:

0

Number of Full Leaves:

0

Depth of Node 200:

-1

Number of Nodes at Depth 2:

6

Tree as List:

[1, 3, 4, 5, 6, 7, 8, 10, 11, 13, 15, 16, 17, 18, 23, 25, 28, 30, 32, 33, 34, 35, 36, 37, 40, 41, 42, 44, 47, 48, 49, 50, 52, 53, 54, 59, 63, 64, 69, 70, 71, 73, 75, 77, 79, 80, 83, 90, 92, 95]

Time to run Program: 0.3290574550628662 seconds

CASE 3:

Size: 100

Height:

3

Smallest at Depth 1 :

9

Largest at Depth 2:

96

Printing at Depth 2:

3 6 12 15 21 24 30 33 39 42 48 51 57 60 66 69 75 78 84 87 93 96

Number of Full Nodes:

0

Number of Full Leaves:

0

Depth of Node 200:

-1

Number of Nodes at Depth:

6

Tree as List:

[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100]

Time to Run program: 1.7063775062561035 seconds

CASE 4:

Size: 1000

Height:

5

Smallest at Depth 1:

151

Largest at Depth 2:

1897

Printing at Depth 2:

54 107 212 269 388 444 539 596 702 774 871 924 1032 1084 1178 1239 1364 1426 1538 1585 1694 1746 1843 1897

Number of Full Nodes:

0

Number of Full Leaves:

0

Depth of Node 200:

-1

Number of Nodes at Depth:

6

Tree as List:

[4, 5, 7, 10, 15, 16, 17, 19, 21, 23, 24, 25, 28, 29, 30, 33, 36, 37, 38, 42, 46, 48, 49, 50, 52, 53, 54, 55, 58, 59, 60, 61, 62, 64, 65, 67, 71, 72, 74, 75, 83, 84, 86, 87, 88, 91, 92, 93, 94, 96, 97, 102, 103, 107, 108, 110, 112, 114, 117, 118, 120, 121, 123, 124, 125, 126, 127, 128, 130, 131, 133, 134, 135, 139, 140, 141, 144, 147, 148, 149, 151, 152, 155, 156, 157, 160, 161, 168, 170, 171, 172, 174, 179, 181, 187, 188, 192, 195, 198, 200, 202, 204, 205, 206, 207, 209, 211, 212, 219, 221, 222, 223, 224, 228, 229, 233, 234, 236, 238, 239, 241, 246, 247, 248, 249, 252, 256, 257, 260, 262, 263, 265, 267, 268, 269, 271, 277, 279, 281, 283, 285, 287, 289, 290, 293, 302, 303, 307, 308, 309, 311, 313, 317, 318, 320, 321, 322, 324, 325, 327, 328, 330, 333, 335, 336, 337, 344, 347, 348, 350, 351, 354, 357, 358, 360, 361, 366, 370, 371, 374, 375, 376, 377, 378, 379, 384, 386, 387, 388, 390, 392, 394, 396, 399, 400, 401, 402, 404, 409, 410, 411, 412, 413, 421, 422, 424, 426, 428, 430, 433, 434, 436, 439, 441, 443, 444, 445, 448, 449, 450, 451, 452, 453, 454, 456, 458, 459, 460, 464, 467, 469, 470, 474, 477, 478, 479, 480, 482, 483, 484, 486, 487, 488, 492, 493, 496, 497, 498, 500, 502, 504, 507, 508, 510, 511, 513, 517, 518, 519, 520, 521, 522, 523, 524, 525, 526, 529, 532, 537, 539, 540, 541, 544, 546, 547, 548, 550, 552, 554, 558, 559, 562, 563, 567, 569, 570, 572, 575, 576, 577, 578, 586, 589, 590, 593, 595, 596, 597, 598, 600, 601, 602, 603, 605, 606, 607, 608, 613, 614, 615, 616, 619, 624, 627, 628, 629, 630, 633, 638, 642, 647, 649, 650, 651, 652, 653, 655, 659, 661, 662, 664, 665, 666, 668, 670, 674, 675, 676, 678, 679, 685, 686, 689, 690, 694, 697, 698, 699, 700, 701, 702, 705, 707, 709, 714, 715, 717, 719, 720, 722, 723, 735, 738, 740, 742, 746, 748, 750, 754, 757, 758, 759, 760, 764, 766, 767, 768, 774, 775, 776, 777, 779, 781, 783, 789, 792, 796, 797, 799, 800, 801, 802, 803, 806, 808, 809, 810, 812, 813, 815, 816, 817, 818, 820, 821, 822, 824, 825, 826, 828, 831, 832, 833, 834, 836, 838, 845, 849, 851, 852, 853, 854, 856, 857, 858, 859, 861, 863, 865, 867, 869, 871, 872, 874, 876, 878, 879, 881, 890, 892, 893, 895, 896, 903, 904, 905, 906, 907, 909, 910, 911, 912, 913, 915, 918, 919, 920, 923, 924, 925, 932, 933, 934, 935, 940, 941, 942, 943, 944, 945, 947, 949, 950, 952, 954, 957, 959, 962, 965, 967, 968, 969, 970, 974, 975, 977, 978, 985, 987, 990, 991, 993, 994, 996, 997, 998, 999, 1000, 1003, 1005, 1006, 1008, 1010, 1011, 1013, 1014, 1016, 1018, 1022, 1023, 1025, 1029, 1032, 1033, 1035, 1036, 1037, 1040, 1042, 1045, 1047, 1048, 1049, 1051, 1054, 1055, 1056, 1060, 1062, 1065, 1067, 1070, 1071, 1073, 1074, 1076, 1077, 1081, 1082, 1084, 1085, 1089, 1090, 1091, 1092, 1093, 1094, 1096, 1098, 1099, 1100, 1101, 1102, 1103, 1104, 1105, 1108, 1109, 1111, 1113, 1114, 1115, 1116, 1117, 1120, 1122, 1123, 1125, 1127, 1129, 1130, 1132, 1135, 1136, 1137, 1140, 1143, 1145, 1146, 1151, 1152, 1153, 1155, 1156, 1161, 1163, 1166, 1167, 1170, 1171, 1173, 1176, 1177, 1178, 1180, 1182, 1183, 1184, 1186, 1187, 1190, 1196, 1197, 1200, 1202, 1203, 1206, 1212, 1214, 1215, 1216, 1218, 1222, 1225, 1226, 1229, 1231, 1233, 1237, 1238, 1239, 1241, 1243, 1244, 1246, 1251, 1254, 1256, 1258, 1260, 1266, 1267, 1268, 1271, 1273, 1275, 1276, 1279, 1280, 1281, 1285, 1286, 1288, 1291, 1294, 1295, 1296, 1298, 1302, 1303, 1307, 1312, 1317, 1321, 1323, 1328, 1329, 1330, 1332, 1336, 1341, 1345, 1346, 1349, 1350, 1351, 1352, 1355, 1356, 1357, 1359, 1361, 1362, 1363, 1364, 1366, 1367, 1369, 1374, 1378, 1381, 1382, 1383, 1385, 1388, 1389, 1390, 1393, 1394, 1400, 1401, 1402, 1403, 1404, 1408, 1410, 1415, 1418, 1419, 1420, 1423, 1426, 1429, 1437, 1438, 1439, 1443, 1445, 1447, 1448, 1451, 1452, 1455, 1462, 1464, 1466, 1468, 1472, 1473, 1474, 1475, 1476, 1477, 1478, 1480, 1482, 1483, 1484, 1489, 1490, 1495, 1497, 1499, 1503, 1505, 1507, 1508, 1509, 1510, 1511, 1513, 1514, 1516, 1517, 1518, 1520, 1522, 1524, 1527, 1528, 1530, 1531, 1532, 1535, 1537, 1538, 1542, 1543, 1544, 1545, 1546, 1547, 1548, 1549, 1552, 1553, 1554, 1560, 1564, 1566, 1569, 1572, 1573, 1575, 1577, 1578, 1579, 1580, 1581, 1582, 1583, 1584, 1585, 1586, 1587, 1588, 1593, 1594, 1598, 1599, 1606, 1608, 1609, 1610, 1612, 1614, 1615, 1621, 1624, 1626, 1628, 1629, 1631, 1632, 1633, 1636, 1639, 1640, 1645, 1646, 1648, 1655, 1656, 1657, 1660, 1661, 1662, 1664, 1666, 1667, 1669, 1672, 1674, 1677, 1678, 1679, 1680, 1681, 1682, 1683, 1684, 1688, 1689, 1690, 1691, 1693, 1694, 1695, 1697, 1699, 1701, 1702, 1704, 1709, 1710, 1712, 1714, 1715, 1718, 1719, 1724, 1725, 1731, 1732, 1734, 1737, 1738, 1739, 1740, 1741, 1742, 1744, 1745, 1746, 1747, 1750, 1751, 1752, 1753, 1755, 1756, 1757, 1758, 1759, 1760, 1761, 1766, 1767, 1772, 1775, 1776, 1778, 1779, 1781, 1782, 1783, 1784, 1786, 1787, 1788, 1789, 1790, 1791, 1793, 1796, 1798, 1800, 1801, 1806, 1807, 1808, 1809, 1810, 1814, 1816, 1819, 1822, 1823, 1826, 1827, 1828, 1829, 1831, 1832, 1836, 1838, 1839, 1843, 1847, 1849, 1850, 1853, 1856, 1857, 1861, 1864, 1865, 1866, 1868, 1870, 1871, 1872, 1873, 1876, 1877, 1878, 1879, 1880, 1883, 1884, 1887, 1888, 1891, 1895, 1897, 1900, 1901, 1904, 1905, 1906, 1907, 1910, 1915, 1916, 1919, 1920, 1922, 1923, 1924, 1926, 1927, 1928, 1929, 1930, 1933, 1934, 1935, 1936, 1937, 1939, 1940, 1942, 1944, 1946, 1948, 1949, 1955, 1956, 1959, 1960, 1961, 1963, 1964, 1966, 1967, 1970, 1973, 1974, 1976, 1977, 1979, 1981, 1982, 1983, 1987, 1988, 1989, 1991, 1998, 1999]

Time to run program: 136.14835166931152

**Time Complexity**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  | | --- | --- | --- | | 25 | 0.026988 |  | | 50 | 0.329057 |  | | 100 | 1.706378 |  | | 1000 | 136.1484 |  | |  |  |  | |
|  |
|  |
|  |
|  |

Therefore, we can decide the that run time of this program is O(n)

**Academic Honesty**

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

Gilbert Velasquez

Gilbert Velasquez

**Source Code:**

|  |
| --- |
| #Gilbert Velasquez |
|  | # CS2302 MW 1:30-2:50 |
|  | # lab 4 |
|  | # Instructor Olac Fuentes |
|  | # TA Anindita Nath and Maliheh Zargaran |
|  | # Date of Last Modification 3/24/2019 |
|  | #The purpose of this lab was to get used to B-Trees. We learned about the 3 types of routes to tackle problems using B-Trees. And given an outline for each type it |
|  | #was easy to convert them to do the desired task once I figured out which type I need to use. In this code there is a method for: Computing the height of a B-Tree, Turning a B-Tree into a sorted list, |
|  | # Getting the smallest element at a given depth, getting the largest element at a given depth, getting the number of nodes at a given depth, printing all the items at a given depth, getting the number of full nodes, |
|  | # getting the number of full leaves, And finding the depth of a given element. This Lab taught me that sometimes the answers arent as complicated as they seem. It also showed me the importance of knowing how to traverse |
|  | # a B\_tree. Knowing how to traverse a B\_Tree was essential for all of these methods. Using the logic gained from BST find smallest and largest is also made simple. |
|  |  |
|  |  |
|  |  |
|  |  |
|  | 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 Height(T): |
|  | if T.isLeaf: |
|  | return 0 |
|  | return 1 + Height(T.child[0]) |
|  |  |
|  | def ExtractTree(T,L): |
|  | if T.isLeaf: #appends every element to list once we reach the leaf |
|  | for t in T.item: |
|  | L.append(t) |
|  | else: |
|  | for i in range(len(T.item)): # Traverses the List in order to get the sorted list |
|  | ExtractTree(T.child[i],L) |
|  | L.append(T.item[i]) |
|  | ExtractTree(T.child[len(T.item)],L) |
|  |  |
|  | def SmallestAtDepth(T,d): #Returns the smallest item at a given depth |
|  | if d==0: |
|  | return T.item[0] #will return the item in index 0 since that is the smallest |
|  | if T.isLeaf: |
|  | return math.inf # returns infinity if we go out of bounds |
|  | return SmallestAtDepth(T.child[0],d-1) # recur until we are at the desired depth |
|  |  |
|  | def LargestAtDepth(T,d): # returns the largest item at a given depth |
|  | if d ==0: |
|  | return T.item[len(T.item)-1] # will return the item in the last index since that is the biggest |
|  | if T.isLeaf: |
|  | return math.inf # returns infinty if we go out of bounds |
|  | return LargestAtDepth(T.child[len(T.item)],d-1) # recur until we are at the desired depth |
|  |  |
|  | def NumNodesAtDepth(T,d): # counts number of nnodes at a given depth |
|  | if d == 0: # if our orignal d is 0 |
|  | return 1 |
|  | if d-1 == 0: # number of nodes at a depth is equal to the number of children from the level above. So we get to that level and return the number of children |
|  | return len(T.child)\*2 |
|  | else: |
|  | for i in range(len(T.child)): |
|  | return NumNodesAtDepth(T.child[i],d-1) |
|  |  |
|  |  |
|  | def PrintAtDepth(T,d): #prints items at a depth |
|  | if d==0: # once we are desired depth print only those nodes |
|  | for i in range(len(T.item)): |
|  | print(T.item[i], end = " ") |
|  | else: |
|  | if not T.isLeaf: #recur with each child and the next depth |
|  | for i in range(len(T.child)): |
|  | PrintAtDepth(T.child[i],d-1) |
|  |  |
|  |  |
|  | def FullNodes(T): # returns number full nodes in a B-Tree |
|  | if len(T.item) == T.max\_items: #if the node is full |
|  | return 1 |
|  | if T.isLeaf: |
|  | return 0 |
|  | else: |
|  | count = 0 # keep track of full nodes |
|  | for i in range(len(T.item)): |
|  | count = count + FullNodes(T.child[i]) #traverse the B-Tree |
|  | return count |
|  |  |
|  |  |
|  | def FullLeaves(T): # returns number of full leaves in a B-Tree |
|  | if len(T.item) == T.max\_items: #check if it's full |
|  | if T.isLeaf: |
|  | return 1 # if it's a leaf then we keep track |
|  | else: |
|  | return 0 # if not we return 0 since it's not full |
|  | else: |
|  | count = 0 |
|  | for i in range(len(T.item)): |
|  | count = count + FullNodes(T.child[i]) #Traverse the B-Tree |
|  | return count |
|  |  |
|  |  |
|  | def FindDepth(T,k): #Returns the Depth of a given key |
|  | if k in T.item: |
|  | return 0 |
|  | if T.isLeaf: # Key wasnt in B-Tree |
|  | return -1 |
|  | if k > T.item[-1]: |
|  | d = FindDepth(T.child[-1],k) #if k is greater than the biggest element in the list go only to the right side |
|  | else: |
|  | for i in range(len(T.item)): |
|  | if k < T.item[i]: |
|  | d = FindDepth(T.child[i],k) |
|  | if d == -1: |
|  | return -1 |
|  | return d + 1 |
|  |  |
|  |  |
|  |  |
|  | L = [30, 50, 10, 20, 60, 70, 100, 40, 90, 80, 110, 120, 1, 11 , 3, 4, 5,200,2,4,5,1,1,2] |
|  | T = BTree() |
|  | for i in L: |
|  | print('Inserting',i) |
|  | Insert(T,i) |
|  | PrintD(T,'') |
|  | #Print(T) |
|  | print('\n####################################') |
|  |  |
|  | print(Height(T)) |
|  | print(SmallestAtDepth(T,1)) |
|  | print(LargestAtDepth(T,2)) |
|  | print(PrintAtDepth(T,2)) |
|  | print(FullNodes(T)) |
|  | print(FullLeaves(T)) |
|  | print(FindDepth(T,200)) |
|  | print(NumNodesAtDepth(T,2)) |
|  | L= [] |
|  | ExtractTree(T,L) |
|  | print(L) |