**Introduction**

The purpose of this assignment was to understand and manipulate B-Trees for the full comprehension of the data structure. The first instruction involves computing the height of the tree, then extracting the items in the B-tree into a sorted list, returning the minimum element in the tree at a given depth d, returning the maximum element in the tree at a given depth d, returning the number of nodes in the tree at a given depth d, printing all the items in the tree at a given depth d, returning the number of nodes in the tree that are full, returning the number of leaves in the tree that are full, and finally, given a key k, return the depth at which it is found in the tree, or return -1 if k is not in the tree.

**Proposed Solution Design and Implementation**

Retrieving the height of a B-tree is one of the simplest tasks to complete as every b-tree is balanced, so producing the height of a tree only requires the counting of the length of only the left or right subtree.

Producing a sorted list from the B-tree was a little more difficult than expected. The reason for the difficulty was because the function produced and submitted retrieves the lists from the lowest depth in order, not the items from left to right: In-Order. To solve this, a second function was created to retrieve the newly created list and then bubble sort it into ascending order.

Returning the smallest element only requires that the function traverses the left children of each subsequent parent then return the first item at the given node when the depth is subtracted to zero. The same can be done for returning the largest item in the tree, except only needing to traverse the right children of each node.

The retrieval of the number of nodes at a given depth is similar to producing the minimum and maximum elements of the b-tree. The main point of the function is to return the length of each list of every given node at the given depth d, this can be done with a for-loop.

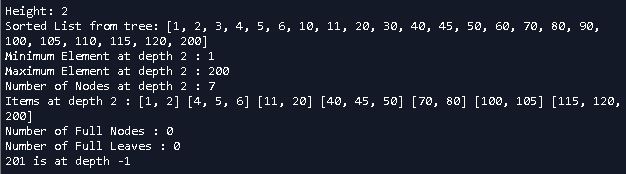
Printing all the items at depth d is very simple. The only task at hand is to use a for-loop to traverse all nodes of the b-tree, then subtracting from d until d equals zero, once this happens, print the list of the current node. Assuming the for-loop ranges from zero to the length of the list minus one, then the printing of each item will be done in order.

To return the number of nodes and leaves that are full, simply traverse the tree and count the number of nodes with lists that are equal to or greater than the variable max\_items.

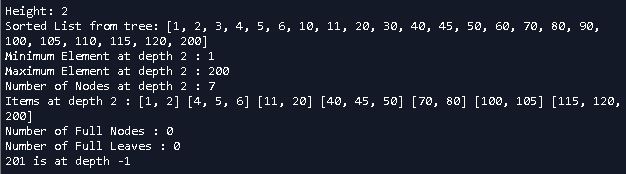
The most difficult task to deal with was the search function. The main goal was to return the depth of an item in the tree, if the item was not present, the method would return -1. This, for the most part was accomplished, however, when searching for an item that is greater than the smallest element in the tree and smaller than the greatest element in the tree, the function returns depth 2, regardless of its presence in the tree.

**Experimental Results**

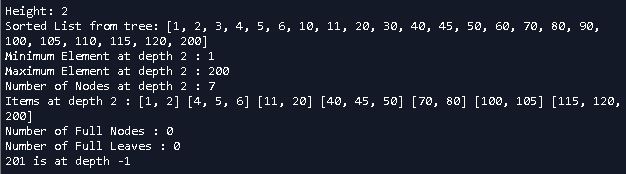
The height of the tree is of length two, this is reflected in the results of the function.

****

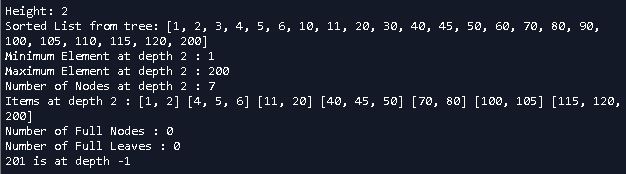
The functions SortedList and Sorter return the newly created list, then sorts it in order.

****

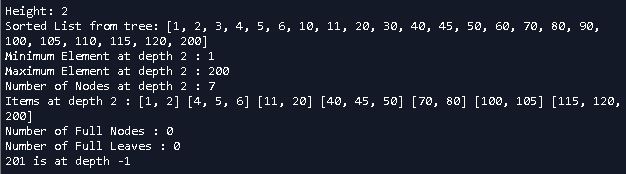
The smallest element in the tree at depth 2 is item 1, the function SmallestAtDepth successfully returns 1 by traversing the furthest-left nodes of the tree and returning the first item in the given list.

****

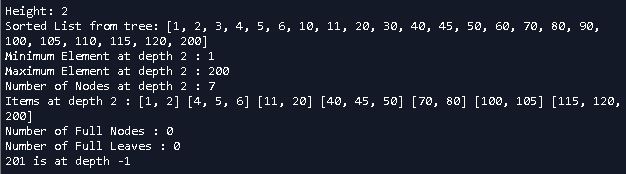
The largest element in the tree at depth 2 is item 200, the function LargestAtDepth successfully returns 200 by traversing the furthest-right nodes of the tree and returning the last item in the given list.

****

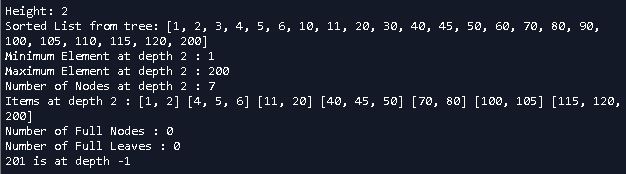
One task from the assignment requires that a function counts the number of nodes at a given depth, NumNodesAtDepth returns 7 since the there are seven lists at depth 2.

****

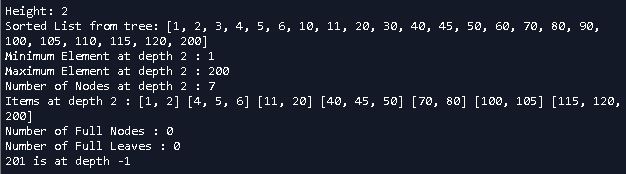
The function PrintItemsAtDepth prints the lists, in ascending order, at depth 2.

****

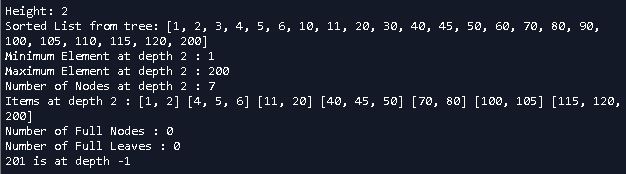
The function FullNodes returns the number of nodes that are equal to or greater than max\_items.

****

The function FullLeaves returns the number of nodes that are equal to or greater than max\_items and are leaves.

****

The function AtDepth traverses the tree to find k, if k is found in the tree, the function returns the depth, else it returns -1 if it is not found. In this case, 201 is not found in the tree, therefore the function returns -1.

****

**Conclusions**

Having completed the assignment, it was understood that this one was the most successfully done. The amount of time personally allocated to completing this assignment was a great amount. This particular data structure, personally, was understood and retained at a higher comprehension level as it was easier to manipulate the lists of the tree. Having accomplished the task at hand fairly well, the most important lesson learned has to deal with figuring out a solution to each problem at hand without over thinking or over complicating the solution.

**Appendix**

|  |
| --- |
| """ |
|  | Course: CS 2302 |
|  | Author: Jacob Montenegro |
|  | Lab: 4 |
|  | Instructor: Dr. Olac Fuentes |
|  | T.A.: Anindita Nath, Maliheh Zargaran |
|  | Date of Last Modification: 3/15/2019 |
|  | Program's Purpose: Understand the process of traversing and manipulating a B-tree. |
|  | """ |
|  |  |
|  | # Code to implement a B-tree |
|  | # Programmed by Olac Fuentes |
|  | # Last modified February 28, 2019 |
|  | import math |
|  |  |
|  | 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 |
|  | c=0 |
|  | for i in T.item: |
|  | if i>k: |
|  | return c |
|  | c +=1 |
|  | 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 function gets the height of the tree |
|  | def height(T): |
|  | #Base Case, just in case the tree doesn't exist |
|  | if T is None: |
|  | return -1 |
|  | #Checks if the current node is a leaf node, which prevents from moving to a NoneType |
|  | if T.isLeaf: |
|  | return 0 |
|  | #counts only one side since this tree is a balanced tree, meaning it will have the same height throughout |
|  | #This statement counts the left subtree. |
|  | return 1 + height(T.child[0]) |
|  |  |
|  | #this function returns a list of the elements from the tree. |
|  | def SortedList(T,L): |
|  | #checks if T exists |
|  | if T is not None: |
|  | #Traverses all branches of the tree, beginning with the left subtree |
|  | for i in range(len(T.child)): |
|  | L = SortedList(T.child[i],L) |
|  | #Concatenates every list from the tree |
|  | L += T.item |
|  | return L |
|  |  |
|  | #This function sorts the list received from the SortedList function |
|  | def Sorter(Q): |
|  | #This is a version of bubble sort that sorts arrays/lists |
|  | done = False |
|  | while done is not True: |
|  | done = True |
|  | for i in range(len(Q)-1): |
|  | #This statement ensures that pointer does not move beyond the scope of the list |
|  | if Q[i+1] == len(Q): |
|  | break |
|  | if Q[i]>Q[i+1]: |
|  | #Swaps from left to right should a preceding element be greater than a succeeding element |
|  | a = Q[i] |
|  | Q[i] = Q[i+1] |
|  | Q[i+1] = a |
|  | done=False |
|  | #returns the sorted list |
|  | return Q |
|  |  |
|  | #Returns smallest item in B-tree |
|  | def SmallestAtDepth(T,d): |
|  | #Base Case: returns -inf if T is none |
|  | if T is None: |
|  | return -math.inf |
|  | #Base Case: if the height of the tree is less than the chosen depth, it returns -inf |
|  | if d > height(T): |
|  | return -math.inf |
|  | #when d is 0, returns the first item of the list at given depth |
|  | if d == 0: |
|  | return T.item[0] |
|  | #returns the left most child, since it would have the smallest elments of the succeeding nodes |
|  | return SmallestAtDepth(T.child[0],d-1) |
|  |  |
|  | #Returns largest item in B-tree |
|  | def LargestAtDepth(T,d): |
|  | #Base Case: returns -inf if T is none |
|  | if T is None: |
|  | return -math.inf |
|  | #Base Case: if the height of the tree is less than the chosen depth, it returns -inf |
|  | if d > height(T): |
|  | return -math.inf |
|  | #when d is 0, returns the last item of the list at given depth |
|  | if d == 0: |
|  | return T.item[len(T.item)-1] |
|  | #returns the right most child, since it would have the largest elments of the succeeding nodes |
|  | return LargestAtDepth(T.child[len(T.item)],d-1) |
|  |  |
|  | #Returns the number of nodes at a given depth |
|  | def NumNodesAtDepth(T,d): |
|  | #If the given T is None, there are no nodes, returns 0 |
|  | if T is None: |
|  | return 0 |
|  | #When d is 0, returns 1 since a node would need to exist here |
|  | if d == 0: |
|  | return 1 |
|  | #initializes a variable to count the number of nodes at a given depth |
|  | count = 0 |
|  | #Traverses the whole tree |
|  | for i in range(len(T.child)): |
|  | #counts number of nodes at depth d |
|  | count += NumNodesAtDepth(T.child[i],d-1) |
|  | #returns number of nodes at depth d |
|  | return count |
|  |  |
|  | #Prints the items at given depth |
|  | def PrintItemsAtDepth(T,d): |
|  | #Base Case to check if T is None or not |
|  | if T is not None: |
|  | #prints the list of integers when d is 0 |
|  | if d == 0: |
|  | print(T.item,end=' ') |
|  | #if d is not 0, traverses tree to every subsequent node |
|  | for i in range(len(T.child)): |
|  | PrintItemsAtDepth(T.child[i],d-1) |
|  |  |
|  | #returns the number of full nodes |
|  | def FullNodes(T): |
|  | #Base Case: returns 0 if T is none since there are no nodes to count |
|  | if T is None: |
|  | return 0 |
|  | #c keeps count of full nodes |
|  | c = 0 |
|  | #checks if the node is full, adds 1 if it is |
|  | if IsFull(T): |
|  | c += 1 |
|  | #Traverses to every other node in tree, adds to c every count |
|  | for i in range(len(T.child)): |
|  | c += FullNodes(T.child[i]) |
|  | #returns the number of full nodes |
|  | return c |
|  |  |
|  | #returns the number of full leaves |
|  | def FullLeaves(T): |
|  | #Base Case: returns 0 if T is none since there are no nodes to count |
|  | if T is None: |
|  | return 0 |
|  | #checks if the node is a leaf, checks if it is full, adds 1 if it is |
|  | if T.isLeaf and IsFull(T): |
|  | return 1 |
|  | #c keeps count of full leaves |
|  | c = 0 |
|  | #Traverses to every other node in tree, this is done to reach all the leaves, adds to c every count |
|  | for i in range(len(T.child)): |
|  | c += FullLeaves(T.child[i]) |
|  | #returns the number of full leaves |
|  | return c |
|  |  |
|  | #Returns the depth of the given item in the tree |
|  | def AtDepth(T,k): |
|  | #returns -1 since there are no depths to check in a NoneType |
|  | if T is None: |
|  | return -1 |
|  | #if item is found, returns 0 |
|  | if k in T.item: |
|  | return 0 |
|  | #adds 1 to depth |
|  | c = 1 |
|  | #try and except are here just in case an exception is thrown |
|  | try: |
|  | #checks right subtree if k is greater than the greatest item of the current node |
|  | if k > T.item[-1]: |
|  | c += AtDepth(T.child[-1],k) |
|  | #checks left subtree if k is less than the smallest item of the current node |
|  | elif k < T.item[0]: |
|  | c += AtDepth(T.child[0],k) |
|  | except: |
|  | #subtracts three, returns -1 if counter goes beyond tree |
|  | c = -3 |
|  | #returns c |
|  | return c |
|  |  |
|  |  |
|  | 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: |
|  | Insert(T,i) |
|  |  |
|  | PrintD(T,'') |
|  | #Print(T) |
|  | print('\n####################################') |
|  |  |
|  | print('Height:',height(T)) |
|  |  |
|  | Q = [] |
|  | Q = SortedList(T,Q) |
|  | Q = Sorter(Q) |
|  | print('Sorted List from tree:',Q) |
|  | d=2 |
|  | print('Minimum Element at depth',d,':',SmallestAtDepth(T,d)) |
|  | print('Maximum Element at depth',d,':',LargestAtDepth(T,d)) |
|  | print('Number of Nodes at depth',d,':',NumNodesAtDepth(T,d)) |
|  |  |
|  | print('Items at depth',d,':',end=' ') |
|  | PrintItemsAtDepth(T,d) |
|  | print() |
|  |  |
|  | print('Number of Full Nodes',':',FullNodes(T)) |
|  | print('Number of Full Leaves',':',FullLeaves(T)) |
|  | k = 201 |
|  | c = 0 |
|  | print(k,'is at depth',AtDepth(T,k)) |

I certify that this project is entirely my own work. I wrote, debugged, and tested the code 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. Signed, Jacob M. Montenegro.