Laboratory #4 B-Trees

Data Structures 2302

Ivan Gastelum

Professor Olac Fuentes

TA: Nath Anindita

**INTRODUCTION:**

This laboratory project’s main purpose is to learn about the basic concepts and functions of a B-Tree using the Python language. A B-Tree is a type of data structure which stores information, more specifically values in an ordered way similar to a Binary Search Tree (BST). The only difference of a B-Tree is that it stores the values in an array which can be referred to as nodes, additionally a B-tree node can have more than 2 children as opposed to those ‘left’ and ‘right’ children. The nodes are array types, and the items inside the nodes can be called keys. The B-Tree also has other attributes such as a maximum keys that those arrays can have. In this laboratory, 9 different function were developed, in addition to some other more basic functions of a B-Tree.

1. Compute the height of the tree

2. Extract the items in the B-tree into a sorted list.

3. Return the minimum element in the tree at a given depth d.

4. Return the maximum element in the tree at a given depth d.

5. Return the number of nodes in the tree at a given depth d.

6. Print all the items in the tree at a given depth d.

7. Return the number of nodes in the tree that are full.

8. Return the number of leaves in the tree that are full.

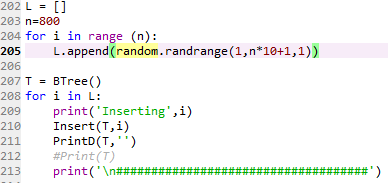
9. Given a key k, return the depth at which it is found in the tree, of -1 if k is not in the tree.

**PROPOSED SOLUTION DESIGN AND IMPLEMENTATION:**

1. Attaining the height or also called the depth of the tree can be simply be a recursive function that keeps track of the number of times a node has been traversed down the B-Tree. Oh notation:
2. Extracting the items in the B-Tree into a sorted list. The function can call an empty list as well as the Tree object ‘T’. In order to traverse a Tree is by recursively calling each children, but since in this case arrays are typically traversed by a simple loop, this function of recursion and a loop can be combined. Oh notation:
3. Returning the minimum value in the B-Tree. This same logic can be done as the one we use in a Binary Search Tree, a balanced and sorted B-tree will have the minimum value to the very first leaf. In this case traversing through the most left side and get to the first left leaf is done recursively with no addition of a loop.
4. Same logic a previous traversal way to get to the expected value. In this case, the direction changes to the most right of the B-Tree. Same Oh notation in time complexity.
5. In this case, since it is necessary to traverse through every children of the traverse array, another loop with a recursive method should be implemented. In this case, it was necessary to only count +1 once we have successfully gotten accessed into a children, and we have gotten to the desired depth called by the method. Worst case Oh notation:
6. Printing all items or keys at a given depth. At this point we need to traverse through the entire total number of children once we hit to the desired depth. In this case, implementing another loop with a recursive call inside is needed. Worst case Oh notation is same as above.
7. Returning the number of full nodes can share a same logic, hence depending on the size of the tree, the running times can be drastically increased.
8. Same logic with the only change of an if condition that checks when a leaf has been traversed in order to count +1 if the node is Full.
9. The search method to find a key element in the Tree can be designed in a way of traversing the whole tree at its worst case scenario, but this would be very inefficient if there exists another way of doing it. By implementing a method which expects to locate the key by obtaining the index at which the

**EXPERIMENTAL RESULTS:**

In order to evaluate the efficiency of the functions in terms of running time complexities, I modified the code to perform a table showing the drastic differences in running times once we append different number of keys into the tree. As the running complexities appear to be enormous with a Oh notation of (2^n) the number of n will be drastically low. As follows, the modified code was done in order to append random numbers into the B-Tree:



With this code I was able to see different running times as I was increasing the value of n. See below charts and tables to analyze.

|  |  |
| --- | --- |
| **Number of items** | **Running Time** |
| 15 | 0 |
| 25 | 31.249 |
| 45 | 559.236 |
| 60 | 1078.093 |
| 100 | 2907.211 |
| 200 | 11665.18 |
| 500 | 69144.31 |
| 600 | 1201698.56 |

As shown in the Table and Graph, it can be shown that once the number of items inserted into the B Tree, its time complexities can be drastically increased by a factor of power of n with base 2 as shown in previous Oh notation analyzes.

**CONCLUSIONS:**

As a result we can conclude that B-Tress just as any other Trees, it can time consuming when inserting various items into it, as it has a huge running time Oh notation. Also, traversing trough the Tre in order to Search, Obtain Minimum or Maximum, Sorting values into a List also have a huge Oh notation that notably be time consuming with lots of data in play.

**APPENDIX:**

