IMPLEMENTATION OF MULTIWAY TREE

SRI KRISHNA SAHOO, 2017MT60786

With this report file is an implementation of multiway tree with variable order of tree. It can insert, delete, search for elements in addition to printing the tree “in” and “pre” order. These traversals are like binary searches except each node may have multiple keys which are printed in ascending order.

It does allow duplicated and when a duplicate is found, it inserts it as a predecessor. When deleting, the element which comes first “in” order is deleted.

Following algorithms from GeeksForGeeks are implemented in the program. Code is my own and not taken from the website.

OPERATIONS IN MULTIWAY TREE:

Search
Search is similar to the search in Binary Search Tree. Let the key to be searched be k. We start from the root and recursively traverse down. For every visited non-leaf node, if the node has the key, we simply return the node. Otherwise, we recur down to the appropriate child (The child which is just before the first greater key) of the node. If we reach a leaf node and don’t find k in the leaf node, we return NULL.

Traverse
Traversal is also similar to traversal of Binary Tree. For inOrder traversal, we start from the leftmost child, recursively print the leftmost child, then repeat the same process for remaining children and keys. In the end, recursively print the rightmost child. In PreOrder Traversal, we recursively print the leftmost child then parent and then rightmost child.

Insertion
1) Initialize x as root.
2) While x is not leaf, do following
..**a)** Find the child of x that is going to be traversed next. Let the child be y.
..**b)** If y is not full, change x to point to y.
..**c)** If y is full, split it and change x to point to one of the two parts of y. If k is smaller than mid key in y, then set x as the first part of y. Else second part of y. When we split y, we move a key from y to its parent x.
3) The loop in step 2 stops when x is leaf. x must have space for 1 extra key as we have been splitting all nodes in advance. So simply insert k to x.

Deletion

1. If the key k is in node x and x is a leaf, delete the key k from x.
2. If the key k is in node x and x is an internal node, do the following.
a) If the child y that precedes k in node x has at least t keys, then find the predecessor k0 of k in the sub-tree rooted at y. Recursively delete k0, and replace k by k0 in x. (We can find k0 and delete it in a single downward pass.)
b) If y has fewer than t keys, then, symmetrically, examine the child z that follows k in node x. If z has at least t keys, then find the successor k0 of k in the subtree rooted at z. Recursively delete k0, and replace k by k0 in x. (We can find k0 and delete it in a single downward pass.)
c) Otherwise, if both y and z have only t-1 keys, merge k and all of z into y, so that x loses both k and the pointer to z, and y now contains 2t-1 keys. Then free z and recursively delete k from y.

3. If the key k is not present in internal node x, determine the root x.c(i) of the appropriate subtree that must contain k, if k is in the tree at all. If x.c(i) has only t-1 keys, execute step 3a or 3b as necessary to

guarantee that we descend to a node containing at least t keys. Then finish by recursing on the appropriate child of x.

a) If x.c(i) has only t-1 keys but has an immediate sibling with at least t keys, give x.c(i) an extra key by moving a key from x down into x.c(i), moving a key from x.c(i) 's immediate left or right sibling up into x, and moving the appropriate child pointer from the sibling into x.c(i).

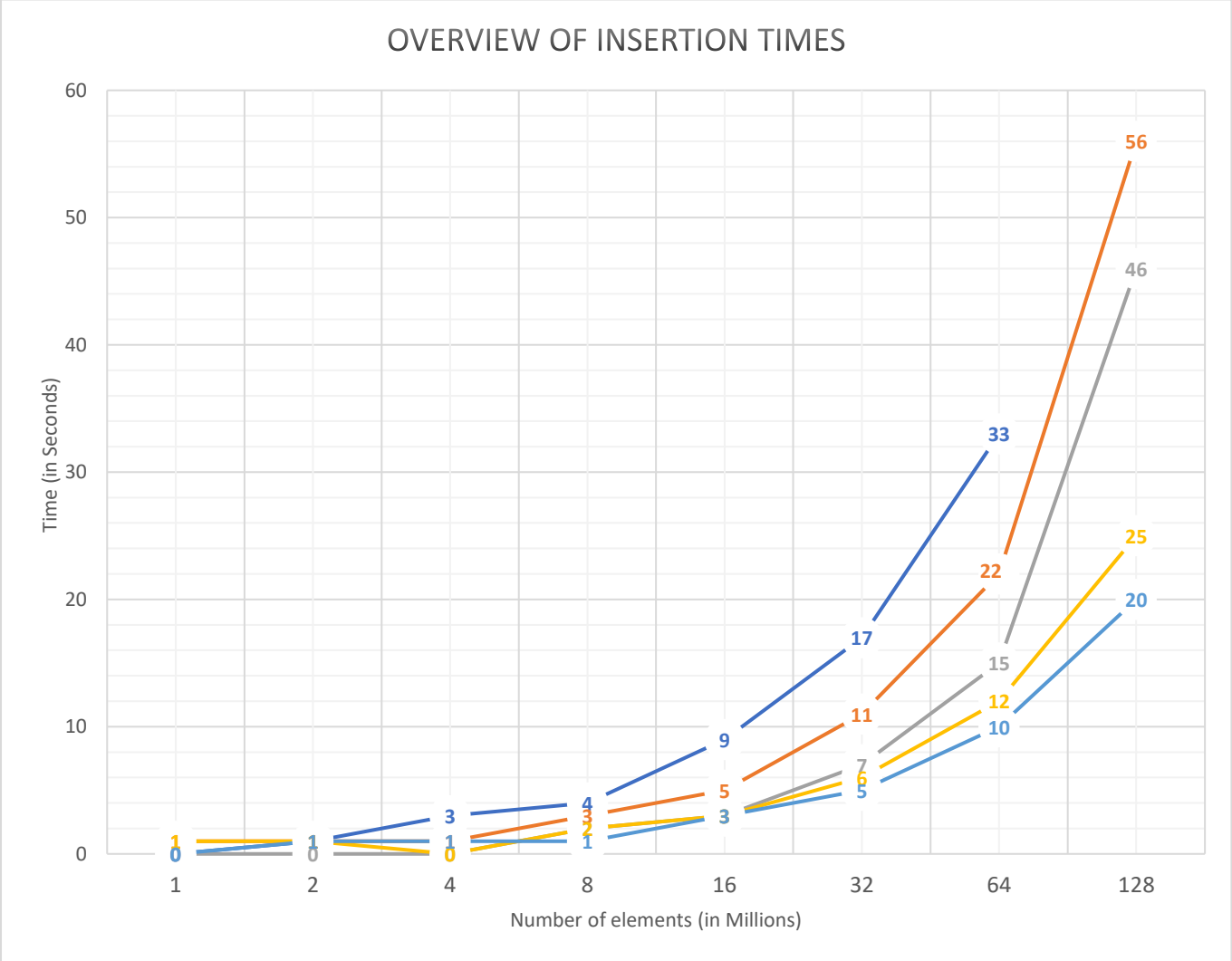
b) If x.c(i) and both of x.c(i)'s immediate siblings have t-1 keys, merge x.c(i) with one sibling, which involves moving a key from x down into the new merged node to become the median key for that node.

ANALYSIS OF INSERTION TIME

Here I have analysed the time taken by the insertions operations and if it follows the theoretical time complexity from analysis. I have chosen the number of nodes such that we can differentiate and compare between times for different degrees of trees. Time taken (in seconds) to insert nodes:-

NO. ELEMENTS (times 1000,000)	MINIMUM DEGREE				
	2	3	4	5	8
1	0	1	0	1	0
2	1	1	0	1	1
4	3	1	0	0	1
8	4	3	2	2	1
16	9	5	3	3	3
32	17	11	7	6	5
64	33	22	15	12	10
128	-	56	46	25	20

(more than 128 M elements were not able to be inserted in my laptop)



Observations:

- Expected time complexity = $\log_{degree} elements$
- For low number of nodes, time taken deviates from expected complexity calculation due to least count of clock being seconds and the role of constant time operations.
- For higher values of number of elements, it can be seen that insertions times for min degree = 4 (grey) are around half as much as those for min degree = 2 (deep-blue). For example,

$$\frac{\log 4}{\log 2} \approx 2 \approx \frac{33}{15} \approx \frac{17}{7}$$

- The time for min degree = 2 are around thrice as much as those for min degree = 8. For example,

$$\frac{\log 8}{\log 4} \approx \frac{3}{2} \approx \frac{7}{5} \approx \frac{15}{10} \approx \frac{46}{20}$$

From the observations, it can be evident that the complexity is as expected.