Rutam Kathale November 16, 2023

CS-1203 Problem Set 4

Collaborators: *none*

Problem 4-1. Caves of Steel (d): Write an algorithm (pseudocode is fine) to convert the first struc- ture (of two levels) to the second (of m levels). Count the total number of link reassignments (i.e., the number of times you change a value in node $->$ pointers to point to a different node). Argue the time complexity.

We can convert the initial structure to a structure of 'm' levels in following steps:
1. We traverse through each node in the original structure.
2. For each node, check if its value is divisible by x because we want multiple levels but fibonacci system doesn't make any sense, we want xth line stop at xth station.
3. If a node's value +1 is divisible by x, we can insert it into the xth linked list in the new structure because this line stops at this line and so it belongs to the list in the station and also show that this line stops here. If we consider node's value, for 0 it'll give error, for 1 it'll be divisible by all but it should stop at every other station and so on... According to our scheme, 0 + 1 = 1 will stop at every station, 1+1 = 2 will stop at every other station, 2+1 = 3 will stop at one in every 3 stations, and so on.
4. (Reassignments) Each time an eligible node is inserted into the linked list, we can initialize and keep track of the counter to count total number of link assignments.
This pseudocode will return our converted structure along with reassignments.

In a nutshell:
1. Create an array of m linked lists to represent the levels.
2. Initialize each level's head pointer to NULL.
3. Traverse original structure until we get NULL.
- determine station for the current line - insert into appropriate linked list - increment reassignment counter.

Our time complexity would be O(mN) where N is total number of nodes in our initial structure.
So, we initialize linked lists and all in O(m) for m levels.

We traverse our original structure in O(N).
Then, we insert nodes into the appropriate level. Let's consider insertion as constant time since its possible to always insert at head (and we're not too concerned about the order). Other operations would be reassignments and a print function, both O(N).
So, overall we have our time complexity as O(mN) we are traversing through the original structure, checking all eligible stations, and inserting for each station

Problem 4-2. Runaround (c): Provide a sequence of 15 integers that maximizes the number of rotations required by the AVL tree during the insertion process. What is the worst-case time complexity of insertion in an AVL tree? Argue why your se- quence meets this bound.

We can consider the sequence: 1217, 1216, 1215, 1214, 1213, 1212, 1211, 1210, 1209, 1208, 1207, 1206, 1205, 1204, 1203 as our sequence to have the worst-case time complexity in an AVL Tree.

Our sequence would result in a skewed left tree, needing rotation at each insertion (worst case.)

The worst-case time complexity of AVL tree insertion is O(log n), where n is the number of nodes in the tree.
So we need a single left rotation at each step.
We have N (=15) insertions.
Our sequence meets the given case because we are guaranteed to result in an unbalanced tree at every insertion since we insert left each time.

AVL Tree is self-balancing that is, it balances itself and performs necessary rotations to do so everytime its structure is changed. Number of rotations needed is proportional to the height of the tree (O(log n)).
So, first we traverse the tree.
Get to the node where we need to insert the new node.
Perform rotations during insertion.

We take O(log n) for traversal considering AVL tree having structure as BST (reducing # nodes by half at each level).
Since the time complexity of rotations itself are based on implementation, and no matter what the implementation, it'll have an algorithmic structure to guarantee balancing of tree in constant operation irrespective of # of nodes. So, we can consider its complexity as O(1).

So we have O(log n) for traversal and O(1) for each rotation in worst case and so complexity for an insertion in worst case is O(log n)

Problem 4-3. The Last Question (a): Using the 15 integer values from the autograder for question 2, draw a B tree and a B+ tree

The autograder inputs are: 74, 55, 2, 61, 51, 70, 70, 53, 47, 50, 58, 23, 28, 6, 11.
Based on these, the B and B+ trees drawn are attached on last page. Please note that the order assumed in the construction is 5.

Problem 4-3. The Last Question (b): Argue the worst-case time complexity for insertion and search in a B tree and a B+ tree.

The worst-case time complexity for insertion in a B-tree is O(log n), where n being # of keys in the tree.
Insertion is similar to insertion in trees in general i.e. first finding the appropriate leaf where the new key should be inserted by going from root to leaf. As we know, B trees are balanced, and so their height would approximately be log n and so, traversal would take O(log n).
Then, we insert new key into the leaf.
Based on # nodes in leaf, we split the node by moving median key to the parent (considering key to be inserted in the calculation). This would take O(1) time since its a very deterministic procedure independent on input.
But we might need multiple splits itself if parent itself is full. In worst case, we'll need a split each level until root. So, we basically process in accordance to height of tree (move up each time until root), and so it would take O(log n).

So, we have the time complexity upper bounded on O(log n) for insertion in B Tree.

The worst-case time complexity for search in a B-tree is also O(log n).
We, considering worst case would have to go to leaf from root, and so, the traversal time would be the time for a search.
Since B trees are balanced, their height is log n (n=#nodes), so depending upon the value to be searched, we would atmost traverse to a leaf in O(log n).

The worst-case time complexity for insertion in a B+ tree is also O(log n). In B+ trees, we store the keys in the leaves contributing extra time to adjust the pointers and get the intermediate key values to the leaves and link all leaf nodes, all of which can be done with constant time (so the complexity would exactly be the same).
Similar to B tree, insertion in B+ tree involves:
First we go from root to leaf and find place to insert node in O(log n).
Insert the key in O(1).
Split the node if needed - this takes O(1) for one split.
Continue splitting assuming all nodes until root are full, in O(log n) for n (height of tree) levels.
This gives us O(log n) as overall worst-case complexity for B+ trees.

Again, searching in B+ trees is O(log n) exactly same as B trees.

We'll need to traverse to leaf in worst case which would be the cost to traverse tree which is log n sincc B+ trees are balanced => O(log n).
Then we search our key in leaf node.
This will be again, upper bounded by O(log n) overall.

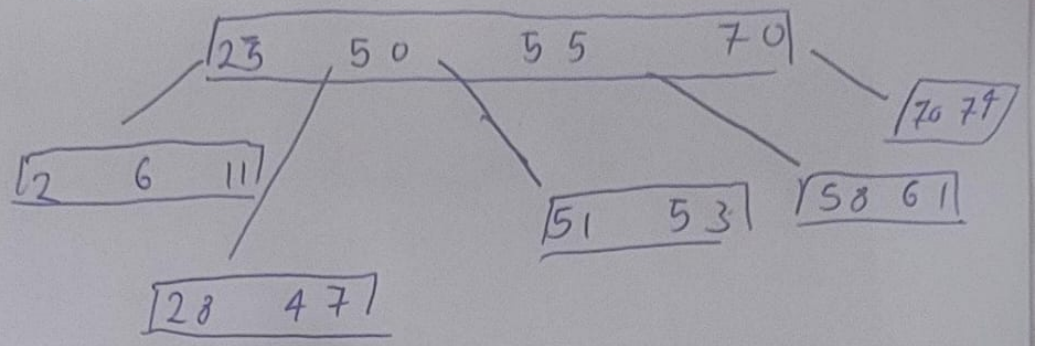
Note: In search for B and B+ trees, we also have to actually find our keys in the node. This depends on the order of the trees. A node can have 2d-1 keys and so, at most we'll spend O(d) time in searching the key in the leaf node. Also, in traversals and splitting until root, the base is 'd' - order of the tree.
This is not included in our analysis on purpose because 'd' is a constant factor throughout our analysis and anyways, our complexity would be upper bounded on dominant operations of traversing the tree and splitting all levels up (both of which are O(log n))).and inclusion of the factor 'd' doesn't help in understanding the growth of the functions based on input.

Problem 4-3. The Last Question (c): For both the B tree and B+ tree, indicate the steps required to construct the tree, including balancing, after every insertion

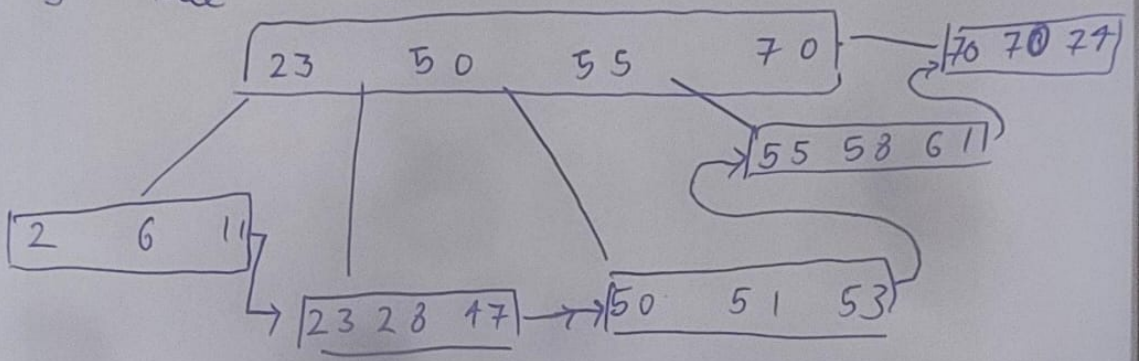
For B tree, the major steps in constructing the tree would be insertion and balancing in every insertion.
So we start with an emoty tree.
Then we insert keys one after the other.
For each insertion, we dind the desired node by traversing from root to leaf (in average case, we'd just stop when we find a child node with its keys in range).
We can insert key in node if it doesn't become full after insertion.
Else, we split the tree by putting the median element on parent node.
Until we get a non-full parent node, we need to keep splitting up one level each time.
We repeat this process until numbers are inserted. From this process, we get a balanced B tree constructed based on our input.

We follow the procedure similarly since B+ trees fundamentally have same structure as B trees apart from the linked list at leaf and the fact that all keys are present in leaf nodes.
So, we start with an empty tree.
We insert keys one by one.
We achieve that by finding the desired node to insert whose keys fall in range of the to-be-inserted key.
If node isn't full, we insert it directly.
Otherwise, we split it by moving the median key up one level.
This is repeated across levels until we get the node which doesn't become full on insertion.
We update the record to point and keep the key in leaf node, and linking it to our linked list for the leaf node.
We continue this until all elements are inserted. We thus result in a balanced B+ tree constructed with our inputs.

B Tree


23 50 55 70
2 6 11
28 47
51 53
58 61
70 74

B + Tree


23 50 55 70
2 6 11
23 28 47
50 51 53
55 58 61
70 70 74