***T- Tree***

T- Tree is an order preserving index structure which is efficient for main memory database management systems. On comparing disk memory and main memory utilization for database operations in main memory prove to be faster and simpler than the disk usage. (Though disks will still be needed to keep a backup of the database). One major advantage that main memory has over disks is that actual attribute values need not be used to maintain an index structure. References (or pointers) are used to do so. They take less space, are easier to use in case of large or complex attributes and their values. A single pointer provides access to the attribute as well as its values. Traversing through the structure and updating becomes tedious with large data to encounter at every go.

Main memory index structures that can be used are arrays, AVL trees, B trees, Chained Bucket Hashing, Linear Hashing and Extendible Hashing. T Tree is a binary tree data structure that has the height balancing nature of an AVL Tree combined with the good update and storage characteristics of the B Tree. Balancing is needed less in comparison to the AVL Tree because a node has greater capacity of storing data.

**Terminologies**

* There are three different types of T-nodes.
* A T-node that has two sub-trees is called an *internal node*.
* A T-node that has only one NIL child pointer is called a *half-leaf node*.
* A T-node that has two NIL child pointers is called a *leaf node*.
* For each internal node A, there is a corresponding leaf (or half-leaf) that holds the data value that is the predecessor to the minimum value in A, and there is also a leaf (or half-leaf) that holds the successor to the maximum value in A.
* The predecessor value is called the *greatest lower bound* of the internal node A, and the successor value is called the *least upper bound* of A.
* For a node N and a value X, if X lies between the minimum element of N and the maximum element of N (inclusive), then we say that node N bounds the value X.
* The data in a T-node is kept in sorted order, its leftmost element is the smallest element in the node and its rightmost element is the largest.

**Figure 1(a) Figure 1(b)**

**Search Algorithm**

* The search always starts from the root of the tree.
* If the search value is less than the minimum value of the node, then search down the sub- tree pointed to by the left- child pointer. Else, if the search value is greater than the maximum value of the node, then search down the sub-tree pointed to by the right-child pointer. Else, search the current node.

**Insertion Algorithm**

* Search for the bounding node.
* If a node is found, then check for room for another entry. If the insert value will fit, then insert it into this node and stop. Else remove the minimum element from the node, insert the original insert value, and make the minimum element the new insert value. Proceed from here directly to the leaf containing the greatest lower bound for the node holding the original insert value. The minimum element (the new insert value) will be inserted into this leaf, becoming the new greatest lower bound value for the node holding the insert value.
* If the search exhausts the tree and no node bounds the insert value, then insert the value into the last node on the search path (which is a leaf or a half-leaf). If the insert value fits then it becomes the new minimum or maximum value for the node. Otherwise, create a new leaf containing only the insert value.
* If a new leaf was added, then check the tree for balance by following the path from the leaf to the root. For each node in the search path (going from leaf to root), if the two sub-trees of a node differ in depth by more than one level, then a rotation must be performed. Once one rotation has been done, the tree is rebalanced and processing stops.

**Deletion Algorithm**

1. Search for the node that bounds the delete value. Search for the delete value within this node. If it is not found report an error and stop.
2. If the delete will not cause an underflow (i.e. if the node has more than the minimum allowable number of entries prior to the delete), then simply delete the value and stop. Else, if this is an internal node, then delete the value and borrow the least upper bound (or the greatest lower bound if the least upper bound doesn’t exist) of this node from a leaf or half-leaf to bring this node’s element count back to the minimum. Else this is a leaf or half leaf so just delete the element. (Leaves are allowed to underflow and the half leaves are dealt with in step 3)
3. If the node is a half leaf and can be merged with a leaf, coalesce the two nodes into one node (a leaf) and discard the other node. Proceed to step 5.
4. If the current node (a leaf) is not empty, then stop. Else free the node and proceed to step 5 to rebalance.
5. If a new leaf was deleted, then check the tree for balance by following the path from the leaf to the root. For each node in the search path (going from leaf to root), if the two sub-trees of a node differ in depth by more than one level, then a rotation must be performed. Once one rotation has been done, the tree is rebalanced and processing stops.

**Rebalancing a T Tree**

**Figure 2(a): LL Rotation**

**Figure 2(b): LR Rotation**

**Figure 2(c): Special T Tree Balancing operations**

Figure 2(a) shows the denotations used. Figure 2(a) and figure 2(b) show T tree balancing operations.

A special rotation as shown in Figure 2(b) is done in case of a T tree because C being a leaf might contain less than the minimum allowable entries per internal node. This is different than a normal AVL rotation. After an LR or RL rotation C becomes an internal node as shown in Regular LR Rotation of Figure 2(b). To resolve this C is first filled with values from its parent so that now it has minimum allowable values. Then the LR or RL rotation is performed as shown in the Special LR Rotation of Figure 2(b).

**Time Complexity**

Let’s say there are n values in the tree and each node holds k values. Then maximum no. of nodes can be (n/k). In that case height will be log (n/k). So,

Time taken to search a bounding node= O (log (n/k)) = O (log n – log k) = O (log n)

In insertion:

* Every rebalancing operation requires maximum log n number of AVL rotations.
* If node doesn’t overflow, time taken= O (k + log n) = O (k + log n)
* If node overflows, time taken= O (k + log n)
* If node is not found= O (1 +log n) = O (log n)

Worst case complexity of insertion= O (k+ log n)

In deletion:

* Every rebalancing operation requires maximum log n number of AVL rotations.
* If node doesn’t underflow, time taken= O (k+ log n)
* Else, if node is internal node, time taken= O ( k/2 + log n) = O (k+ log n)
* If node is a half leaf, time taken= O ( k + log n)
* If node is a leaf node, and becomes empty, time taken= O (log n)

Worst case complexity of deletion= O (k+ log n)

**References**

* Tobin J. Lehman and Michael J. Carey,” A Study of Index Structures for Main Memory Database Management Systems”. VLDB 1986.
* https://en.wikipedia.org/wiki/T-tree