DATA STRUCTURES

B-TREE & K-D TREE

**Time complexity analysis:**

**For K-D Tree:**

**Insertion (insert function):**

* The insertion operation in a k-d tree involves traversing down the tree to find the appropriate position to insert the new node.
* Since the tree is balanced on both dimensions alternately, the average case time complexity for insertion is O(log N), where N is the number of nodes in the tree.
* Best case : O(log N), Worst case : O(N)

**Deletion (del function):**

* Similar to insertion, deletion also requires traversal down the tree to find the node to be deleted.
* Hence, the time complexity for deletion is also O(log N) in the average case.
* Best case : O(log N) , Worst case : O(N)

**Traversal (traverse function):**

* Traversal visits each node in the tree once. Since there are N nodes in the tree, the time complexity of traversal is O(N) in all cases.

**Search (search function):**

* Searching in a k-d tree also involves traversing down the tree to find the target node.
* In the average case, the time complexity of search is O(log N).
* Best case : O(log N), Worst case : O(N)

**Nearest Neighbor Search:**

* Best Case: O(log N) - Similar to search, if the tree is balanced, the nearest neighbor search follows a logarithmic path.
* Worst Case: O(N) - In the worst case, the search may require traversing the entire tree, resulting in linear time complexity.
* Average Case: O(log N) - Assuming a balanced tree, the average time complexity for nearest neighbor search is logarithmic.

**Printing 2D (twodprint function):**

* Printing the 2D representation of the tree involves traversing the entire tree once.
* Therefore, the time complexity of printing 2D is O(N).

**Overall:**

* The average-case time complexity of insertion, deletion, and search operations in the provided k-d tree implementation is O(log N), where N is the number of nodes in the tree.
* Traversal and printing operations have a time complexity of O(N) since they visit each node once.

**Applications of K-D Tree:**

**Nearest Neighbour Search:**

* One of the most common applications of k-d trees is in efficiently finding the nearest neighbor(s) to a given point in multidimensional space.
* This is particularly useful in spatial databases, computer graphics, pattern recognition, and various machine learning algorithms.

**Range Search:**

* K-d trees are also useful for performing range searches, where you need to find all points within a certain distance from a given query point.
* This application is relevant in geographical information systems (GIS), collision detection, and data mining.

**Nearest Neighbour Search implementation:**

**Pseudo-code:**

function nearestNeighborSearch(tree, queryPoint):

bestDistance = INFINITY

bestPoint = NULL

nearestNeighborHelper(tree.root, queryPoint, bestDistance, bestPoint)

return bestPoint

function nearestNeighborHelper(node, queryPoint, bestDistance, bestPoint):

if node is NULL:

return

currentDistance = distance(queryPoint, node.point)

if currentDistance < bestDistance:

bestDistance = currentDistance

bestPoint = node.point

if queryPoint[node.dimension] < node.point[node.dimension]:

nearestNeighborHelper(node.leftChild, queryPoint, bestDistance, bestPoint)

if queryPoint[node.dimension] + bestDistance >= node.point[node.dimension]:

nearestNeighborHelper(node.rightChild, queryPoint, bestDistance, bestPoint)

else:

nearestNeighborHelper(node.rightChild, queryPoint, bestDistance, bestPoint)

if queryPoint[node.dimension] - bestDistance <= node.point[node.dimension]:

nearestNeighborHelper(node.leftChild, queryPoint, bestDistance, bestPoint)

**Time complexity analysis:**

**For B-Tree:**

**insert:**

* In the worst-case scenario, when the root node is full and needs to be split, the time complexity is O(log n), where t is the degree of the B-tree and n is the number of keys in the tree.
* This is because the height of the B-tree increases by 1 when the root splits, and each level can accommodate t keys.

**search:**

* The time complexity for searching in a B-tree is O(log n), where t is the degree of the B-tree and n is the number of keys in the tree.
* This is because B-trees are balanced trees, and at each level, we can eliminate a portion of keys based on their ranges.

**traverse:**

* The time complexity of traversing a B-tree and printing its keys in order is linear with respect to the number of keys in the tree.
* Therefore, the time complexity isO(n), where n is the number of keys in the tree.

**print:**

* The time complexity of printing a B-tree in 2D format is also linear with respect to the number of keys in the tree.
* Therefore, the time complexity isO(n), where n is the number of keys in the tree.

**Applications of B-Tree:**

**Databases:**

* B-trees are extensively used in database systems to index large volumes of data efficiently.
* They help in performing rapid searches, inserts, updates, and deletions on disk-based storage systems.
* B-trees enable efficient range queries and support multi-level indexing in database management systems (DBMS).

**File Systems:**

* B-trees are employed in file systems to manage disk blocks and directories efficiently.
* They provide fast access to files and directories by maintaining an index structure on disk.
* B-trees help in minimizing the number of disk reads and writes required to access or modify files, improving the overall performance of file systems.

**Resolving issues between developers and testers**:

* The delete function in K-D Tree was initially void function and delete did not work and got garbage value in node deleted and to rectify it the function was made to return the updated root value and then worked correctly.
* We couldn’t implement delete operation in B-Tree.