# **Design Of PS04 Problem Bio-Bubble**

**The Data Structure Model Used:**

As requested in the problem statement we have used the Binary Search Tree ADT model to implement the data structure for Bio-Bubble Problem, due to its simple implementation, memory efficiency, automatic searching and efficient searching and inserting.

The BST is a tree-based data structure where each node has at most two children, and the left subtree of a node contains only nodes with keys less than the node's key, and the right subtree only nodes with keys greater than the node's key.

**Function wise Asymptotic notation for time complexity:**

**insert:** This function inserts the record by reading it from input file line by line and creates nodes as per recordSwipeRec function.The time complexity of the insertion operation in a binary search tree is O(log n) in the average case

**recordSwipeRec:** This function records the swipe by reading the numbers on each line creating new node or incrementing the counter variable if node already exist. If there are n swipes in the file, and each insertion operation takes O(log n) (average case )overall time complexity of reading swipes from the file would be O(n log n).

**getSwipeRec:** This function calculates the list of total players but traversing the left tree and right tree and adding the count together and returning it.This function gets the details through traversing the entire tree, visiting each node once. The time complexity is O(n)

**onPremisesRec :** This function checks the player who is still onPremises by getting the counter attribute of the node and dividing it by 2 so if the remainder is 1 the player is still on Premises. Similarly traverse entire tree so Time Complexity is O(n)

**checkEmpRec:** This function checks players history and returns the emp id alng with its counter value and whether inside or outside hotel. The time complexity is O(log n) in the average case for searching in a balanced tree. In the worst case (unbalanced tree), it can be O(n)

**frequentVisitorRec:** This function takes a frequency as input and checks for players counter which is greate or equal to it and returns it .The time complexity is O(n) because the function traverses the entire tree to find players with a frequency greater than a given value.

**printRangePresent:** This function prints the range of Player ID within the given range O(n) complexity

**Overall Complexity :** The most significant factor is the reading of swipes from the file, which has a time complexity of O(n log n).The overall time complexity of the entire program is dominated by the reading of swipes from the file, making the overall time complexity O(n log n) in the average case and O(n^2) in the worst case if the tree becomes unbalanced.

**Alternate Way of Modelling :**

We could implement the data structure using **hashtable** and linked list to store and retrieve player swipe data.

**Hashtable:**

* Use a hashtable to store player information.
* The key in the hashtable would be the player ID, and the corresponding value would be a data structure (e.g., a dictionary) that stores information about the player's swipes.
* The data structure associated with each player ID could include attributes like attrCtr (attribute counter), keeping track of the number of times a player has entered the hotel.

Hashtable operations, including insertion, retrieval, and updates, have an average time complexity of O(1), making them highly efficient. Memory usage may be more efficient compared to a binary search tree, especially for large datasets.

Disadvantage being traversal is harder and order preservation is not as efficient as ADT.