AVL tree

A special AVL tree has been used that processes a data structure which specifies the key and the corresponding ID’s which reference that key. The AVL tree is known to have the following properties:

1. Search [log2n]
2. Insert [log2n]
3. Delete [log2n]

This is because the AVL is a balanced binary tree; it ensures that it is balanced by restructuring itself as items are being inserted and deleted. This is very suitable storage mechanism for the task at hand; especially as the database becomes very large. The speed of retrieving such information is absolutely necessary for large systems. This is very good for search engine optimization which could link up information by searching using a query; and therefore suitable for this task. An alternative structure would be to use a map [hash-table]; however, there are other constraints that may be incurred because of the resource requirements [double the storage].

Vector

A vector was used to store general information with regards to the idea objects. Insertion is O(1) and accessing the data is fast because it is designed to be held sequentially [without incurring any penalties of O(n) transfer]. The sequential design has allowed us to use a binary search on a randomly accessed data structure. In total this would make insertion bigO(1) and search bigO(log2n). There is also no fixed size with vector, we can just keep adding more to the list. It would be more suitable to use this system because of the speed in which you can insert items on the vector.

1. Insertion [O(1)]
2. Deletion [O(n)]
3. Binary Search [O(log2n)]

.