# Data Structures Project Report


## Group Members:

1) **Azmeer Sohail Khan (21i-2977)**
2) **Hamza Shahid (21i-0815)**
3) **Talal Habshi (21i-2456)**


We had to basically create a data base system in this project using different type of trees. We used AVL, red black and B trees for this project. We did insertion searching and other implementations mentioned in the project using these three trees.

AVL trees are a type of self-balancing binary search tree, which means that they maintain their balance (i.e. the height of the left and right subtrees of any node differ by at most one) by rotating the tree as necessary when inserting or deleting nodes. This allows for fast search, insertion, and deletion operations, which makes AVL trees a useful data structure for many applications.

Red-black trees are another type of self-balancing binary search tree. Like AVL trees, they maintain their balance by rotating the tree as necessary when inserting or deleting nodes. However, red-black trees use a different set of rules to determine when and how to rotate the tree, which makes them slightly easier to implement than AVL trees.

B-trees are a type of tree data structure that is optimized for use on disk. They are often used in database systems because they allow for efficient search, insertion, and deletion operations on large amounts of data that cannot be stored entirely in memory. B-trees are similar to binary search trees, but instead of having a maximum of two children per node, they can have a variable number of children, which makes them more space efficient and better suited for use on disk.

To search for a value in an AVL tree, you would first start at the root node of the tree. If the value you are looking for is less than the value at the root node, you would go to the left child of the root node. If the value you are looking for is greater than the value at the root node, you would go to the right child of the root node. This process is repeated until you find the value you are looking for, or until you reach a leaf node (i.e. a node with no children) that does not contain the value you are looking for.

To insert a value into an AVL tree, you would first follow the same process as searching for a value. When you reach the leaf node where the new value should be inserted, you would add the new value as a child of that node. However, because AVL trees are self-balancing, you would then need to check the balance of the tree and rotate it as necessary to maintain the balance of the tree. This process is repeated until the tree is balanced.

To search for a value in a red-black tree, the process is similar to searching in an AVL tree. You would start at the root node and then follow the left or right child depending on whether the value you are looking for is less than or greater than the value at the current node. This process is repeated until you find the value you are looking for, or until you reach a leaf node that does not contain the value you are looking for.

To insert a value into a red-black tree, you would again follow the same process as searching for a value. When you reach the leaf node where the new value should be inserted, you would add the new value as a child of that node. However, because red-black trees are self-balancing, you would then need to update the colors of the nodes and rotate the tree as necessary to maintain the balance of the tree. This process is repeated until the tree is balanced.

To search for a value in a B-tree, you would start at the root node and then follow the appropriate child node according to the value you are looking for. B-trees are similar to binary search trees, but because they can have a variable number of children, the process of following child nodes is slightly more complex. Essentially, you would need to find the child node that contains the range of values that includes the value you are looking for, and then repeat the process until you find the value or reach a leaf node that does not contain the value you are looking for.

To insert a new key into a B-tree, follow these steps. Traverse the tree to find the leaf node where the new key should be inserted. If the leaf node has room, insert the new key and update the pointers as needed. If the leaf node is full, split it and move the middle key to the parent node. Repeat steps 3 and 4 for the parent node as needed until the tree is properly balanced.

We have thoroughly researched the topic, collected and analyzed the data, and presented our findings in a clear and organized manner. We have also discussed the implications of our results and provided suggestions for future research on the topic. We believe that we have covered every aspect of the project and are ready to present it in the demo.