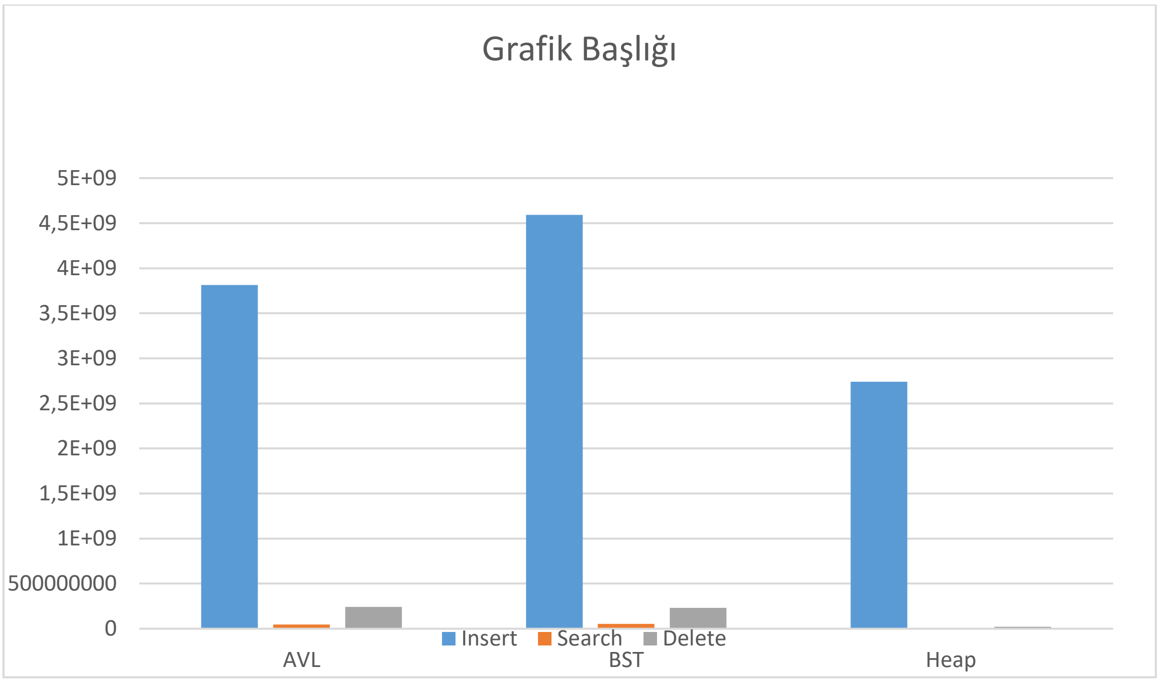
REPORT

Görhem Göktürk SAKA

130144046

Execution Times



BST - Insertion

Insertion begins as a search would begin; if the key is not equal to that of the root, we search the left or right subtrees as before. Eventually, we will reach an external node and add the new key-value pair as its right or left child, depending on the node's key. In other words, we examine the root and recursively insert the new node to the left subtree if its key is less than that of the root, or the right subtree if its key is greater than or equal to the root.

Insert operation requires time proportional to the height of the tree in the worst case, which is O(log n) time in the average case over all trees, but O(n) time in the worst case.

BST - Search

We begin by examining the root node. If the tree is *null*, the key we are searching for does not exist in the tree. Otherwise, if the key equals that of the root, the search is successful and we return the node. If the key is less than that of the root, we search the left subtree. Similarly, if the key is greater than that of the root, we search the right subtree. This process is repeated until the key is found or the remaining subtree is *null*. If the searched key is not found after a *null* subtree is reached, then the key is not present in the tree.

Search operation requires time proportional to the height of the tree in the worst case, which is O(log n) time in the average case over all trees, but O(n) time in the worst case.

BST- Delete

There are three possible cases to consider:

- Deleting a node with no children: simply remove the node from the tree.
- Deleting a node with one child: remove the node and replace it with its child.
- Deleting a node with two children: call the node to be deleted *D*. Do not delete *D*. Instead, choose either its in-order predecessor node or its in-order successor node as replacement node *E* Copy the values of *E* to *D*.If *E* does not have a child simply remove *E* from its previous parent *G*. If *E* has a child, say *F*, it is a right child. Replace *E* with *F* at *E*'s parent.

In all cases, when D happens to be the root, make the replacement node root again.

Broadly speaking, nodes with children are harder to delete. As with all binary trees, a node's in-order successor is its right subtree's left-most child, and a node's in-order predecessor is the left subtree's right-most child. In either case, this node will have only one or no child at all. Delete it according to one of the two simpler cases above.

Heap – Insert

1. Add the element to the bottom level of the heap.
2. Compare the added element with its parent; if they are in the correct order, stop.
3. If not, swap the element with its parent and return to the previous step.

The number of operations required depends only on the number of levels the new element must rise to satisfy the heap property, thus the insertion operation has a worst-case time complexity of O(log *n*) but an average-case complexity of O(1).

Heap – Delete

1. Replace the root of the heap with the last element on the last level.
2. Compare the new root with its children; if they are in the correct order, stop.
3. If not, swap the element with one of its children and return to the previous step. (Swap with its smaller child in a min-heap and its larger child in a max-heap.)

In the worst case, the new root has to be swapped with its child on each level until it reaches the bottom level of the heap, meaning that the delete operation has a time complexity relative to the height of the tree, or O(log *n*).

AVL – Insert

First we add element to the place which is founded by bst-insertion way after that if the new height makes the tree unbalanced, using left or right rotation methods make it balanced again. Complexity of insertion is O(logn).

AVL- Search

The searching way is same as BST-search. But AVL tree is balanced, so complexity of searching is O(logn) in any case.

AVL-Delete

First we delete element to the place which is founded by bst-insertion way after that if the new height makes the tree unbalanced, using left or right rotation methods make it balanced again. Complexity of deletion is O(logn).