B+ Tree v/s AVL

INTRODUCTION

An AVL tree is a special type of binary tree that is always "partially" balanced. The criteria that is used to determine the "level" of "balancedness" is the difference in which the difference between the height of the right and left subtrees or the root node is never more than one. The basic operations of an AVL tree generally involve carrying out the same actions as would a simple binary tree has eg. Insertion,find etc.

But, while inserting, we have to keep the track of heights of right subtree and left subtree and reorder the tree accordingly.

The solution is to dynamically re balance the search tree during insert
or search operations. We have to be careful not to destroy the ordering
invariant of the tree while we re balance. Because of the importance of bi-
nary search trees, researchers have developed many different algorithms
for keeping trees in balance, such as AVL trees, red/black trees, splay trees,
or randomized binary search trees. They differ in the invariants they maintain
(in addition to the ordering invariant), and when and how the re balancing is done.

A B+ tree is an n-nary tree with a variable but often large number of children per node. A B+ tree consists of a root, internal nodes and leaves. The root may be either leaf or node with two or more children.

A B+ tree can be viewed as a B-tree in which each node contains only keys (not pairs), and to which an additional level is added at the bottom with linked leaves.

The primary value of a B+ tree is in storing data for efficient retrieval in a block-oriented storage context—in particular, file systems.

Order of a B + -Tree

• The *order* of a B + -Tree is the number of keys and pointers that an internal node can contain. An order size of *m* means that an internal node can contain *m-1* keys and *m* pointers.

• The order size is important because it determines how large a B + -Tree will become.

• For example, if the order size is small then fewer keys and pointers can be placed in one node and so more nodes will be required to store the index.
If the order size is large then more keys and pointers can be placed in a node and so fewer nodes are required to store the index.

B+ Trees

Operations:

1. Search:

The root of a B^+ Tree represents the whole range of values in the tree, where every internal node a subinterval.

We are looking for a value k in the B^+ Tree. Starting from the root, we are looking for the leaf which may contain the value k. At each node, we figure out which internal pointer we should follow. An internal B^+ Tree node has at most $d \leq b$ children, where every one of them represents a different sub-interval. We select the corresponding node by searching on the key values of the node.

```
Function: search (k)
   return tree_search (k, root);

Function: tree_search (k, node)
   if node is a leaf then
      return node;
   switch k do
      case k < k_0
         return tree_search(k, p_0);
      case k_i ≤ k < k_{i+1}
         return tree_search(k, p_i);
      case k_d ≤ k
         return tree_search(k, p_d);
```

2. **Insertion:**

Perform a search to determine what bucket the new record should go into.

- •If the bucket is not full (at most b - 1 entries after the insertion), add the record.
- •Otherwise, split the bucket.
    - •Allocate new leaf and move half the bucket's elements to the new bucket.
    - •Insert the new leaf's smallest key and address into the parent.
    - •If the parent is full, split it too.
        - •Add the middle key to the parent node.
    - •Repeat until a parent is found that need not split.
- •If the root splits, create a new root which has one key and two pointers.

B-trees grow at the root and not at the leaves.

3. **Deletion:**

- •Start at root, find leaf L where entry belongs.
- •Remove the entry.
    - •If L is at least half-full, done!
    - •If L has fewer entries than it should,
        - •Try to re-distribute, borrowing from sibling (adjacent node with same parent as L).
        - •If re-distribution fails, merge L and sibling.
- •If merge occurred, must delete entry (pointing to L or sibling) from parent of L.
- •Merge could propagate to root, decreasing height.

Comparison Between AVL Trees And B+ Trees

B + tree is used when you're managing more than thousands of items and you're paging them from a disk or some slow storage medium.

AVL tree is used when your inserts and deletes are infrequent relative to your retrievals.

Even when virtual memory isn't an issue, cache-friendliness often is, and B+ trees are particularly good for sequential access - the same asymptotic performance as a linked list, but with cache-friendliness close to a simple array. All this have O(log n) search, insert and delete.

B+ trees do have problems, though - such as the items moving around within nodes when you do inserts/deletes, invalidating pointers to those items.

Efficiency of B+ trees:

In B+-tree, the I/O cost of insertion and deletion is O(log n (base b)) where b is the order of the tree.
The insert,delete and search algorithms examine a single path from root to leaf. Asymptotically the complexity of these algorithms are equal to the height of the tree. Since there are n objects, and the minimum fan-out of the tree is B, the height of the tree is O(log n(base b)). So the complexity of the algorithms is O(log n(base b)) as well.

Time Complexity

Searching:

Interior nodes a B+-Tree of order d have from d to 2d children.
Let F(n) be the timing function for finding a key in a B+-Tree of order d. Let n be the number of unique keys in the B+-Tree.

The time required to find a key is the product of the time required to search a node, N (d) , multiplied by the number of nodes in the path from the root to a leaf (h + 1) . The number of nodes in the path from the root to a leaf is h+1 where h is the height of the B+-Tree.

F (n) = O (N (d)(h + 1))

A binary search is used to search a particular node. A node in a particular tree contains at most 2d keys.
N (d) = O(log 2d(base 2)) = O(log d(base 2) + 1)

Unique keys are contained the leaves of a B - Tree. Interior nodes of the B+-Tree having the greatest height have d children and d keys in the l leaves. There are d nodes at level l. Let n be the number of unique keys in a B+-Tree of order d and height h.
n = d^h.d = d^(h+1)
h + 1 = log d n
Thus
F (n) = O(log n(base d)(log d(base 2) + 1))
F (n) = O(log n(base d) log (2d)(base 2))

Insertion:

In the worst case, a key is inserted into every node on the path to where the new key is inserted. Inserting the new key causes a chain reaction. Every node on the path to the new key is split and a parent hoisted to the next higher level. The length of the path from root to leaf is h, the height of the tree. As a result of inserting a new key, the height of the tree could be increased by one. Let n be the number of unique keys in the B+-tree after the new key is inserted.

The number of nodes for which a new key can be inserted is one more than the

height of the tree.
h + 1 = log n(base d)

The time required to insert a node is O (2d)
I (n) = O(2d log n(base d))

The time required to delete a node from a B+-tree is the same as the time required to insert a node.

AVL Trees

Time complexity:

Since, the height of AVL trees always remain close to log n(base 2); therefore,
insertion,deletion and search all require O(log n(base 2)) time taking rebalancing time as constant.

Operations:-

1. **Insert:**

A binary search tree T is called if for every node v, the height of v's children differ by at most one.

Inserting a node into an AVL tree involves performing an expandExternal(w) on T, which changes the heights of some of the nodes in T.

If an insertion causes T to become unbalance we travel up the tree from the newly created node until we find the first node x such that its grandparent z is unbalanced node.

Since z became unbalanced by an insertion in the subtree rooted at its child y, height(y) = height(sibling(y)) + 2

To re balance the subtree rooted at z, we must perform a restructuring

we rename x, y, and z to a, b, and c based on the order of the nodes in an in-order traversal.

z is replaced by b, whose children are now a and c whose children, in turn, consist of the four other subtrees formerly children of x, y, and z.

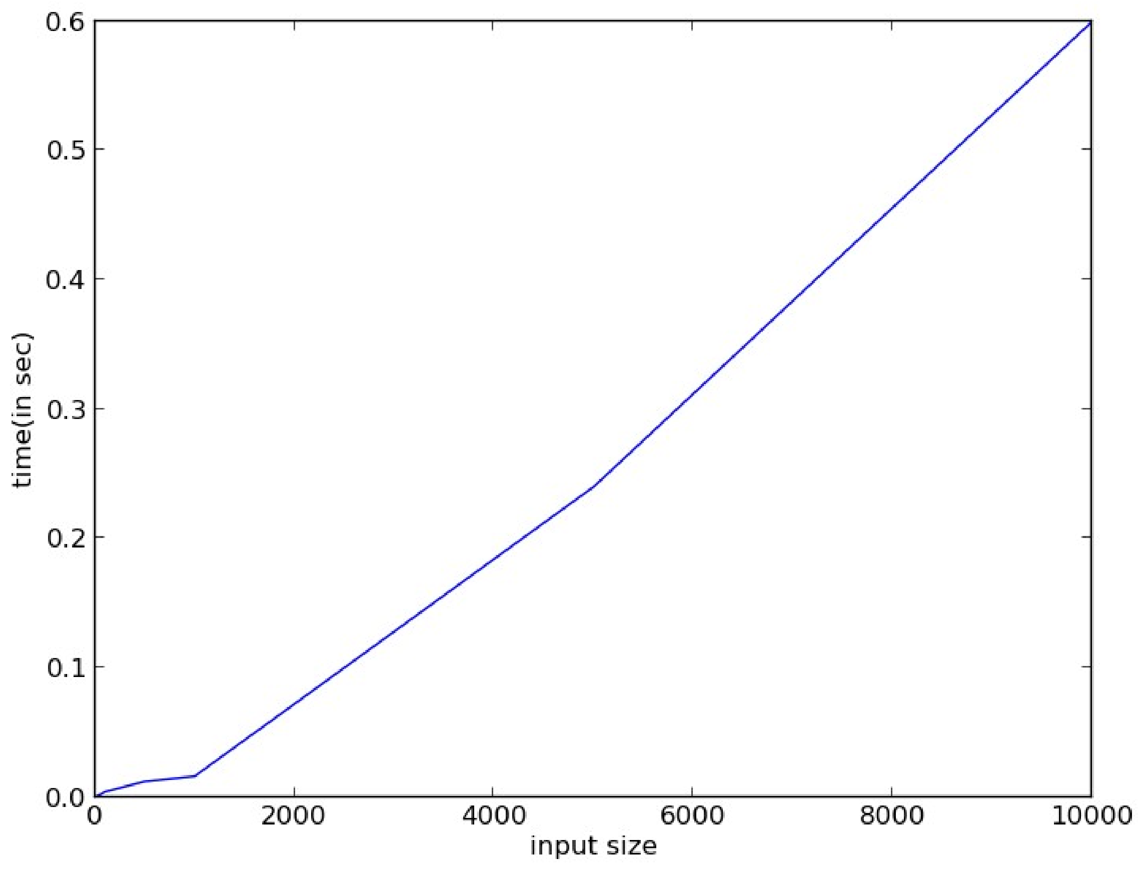
2. **Deletion:**
   Reduce the problem to the case when the node x to be deleted has at most one child.
   Delete x. We use a flag to show if the height of a subtree has been shortened.
   While flag is 1 do the following steps
   for each node p on the path from the parent of x to the root of the tree.
   When flag becomes 0, the algorithm terminates.

Case 1: Node p has balance factor equal.
Case 2: The balance factor of p is not equal, and the taller subtree was shortened.
Case 3: The balance factor of p is not equal, and the shorter subtree was shortened. Apply a rotation as in case of insertion.

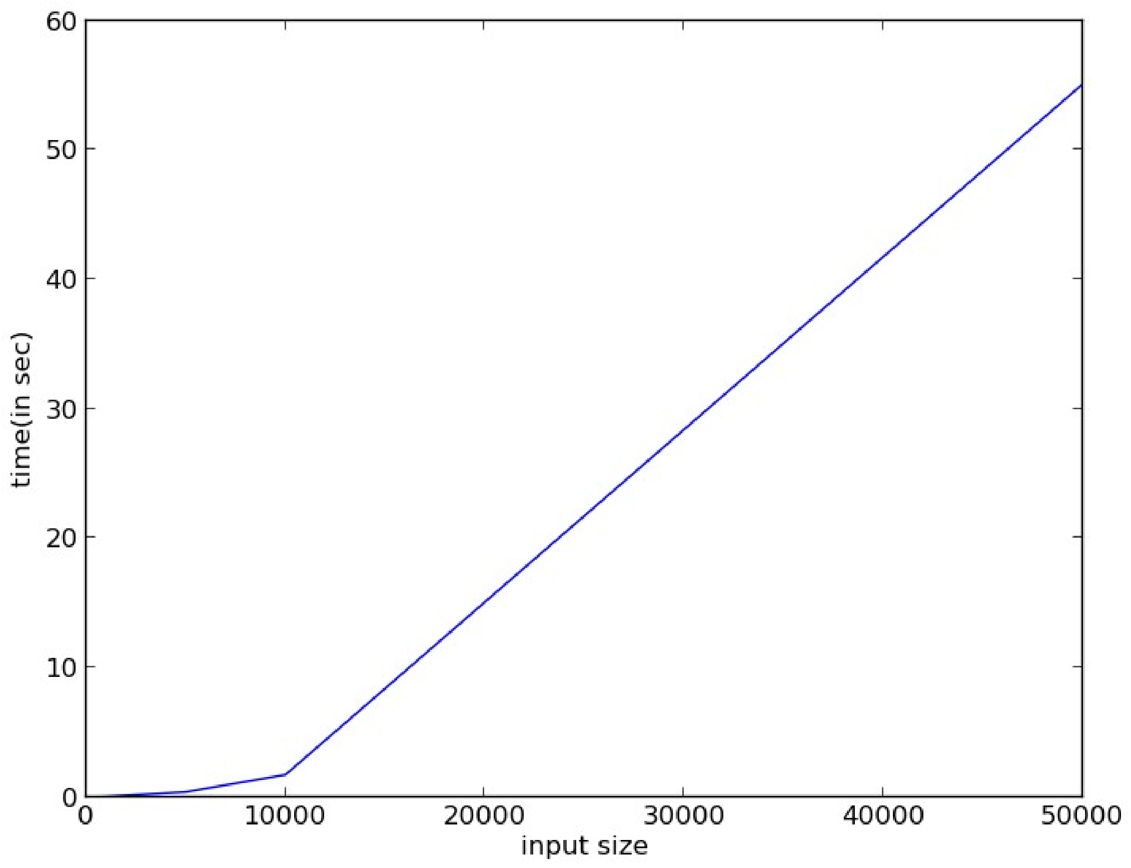
3. **Search:**
   This includes normal searching as in case of binary search trees.
   If the value to be searched is less than the root, we search in left subtree recursively , else search in right subtree recursively.

Graphs

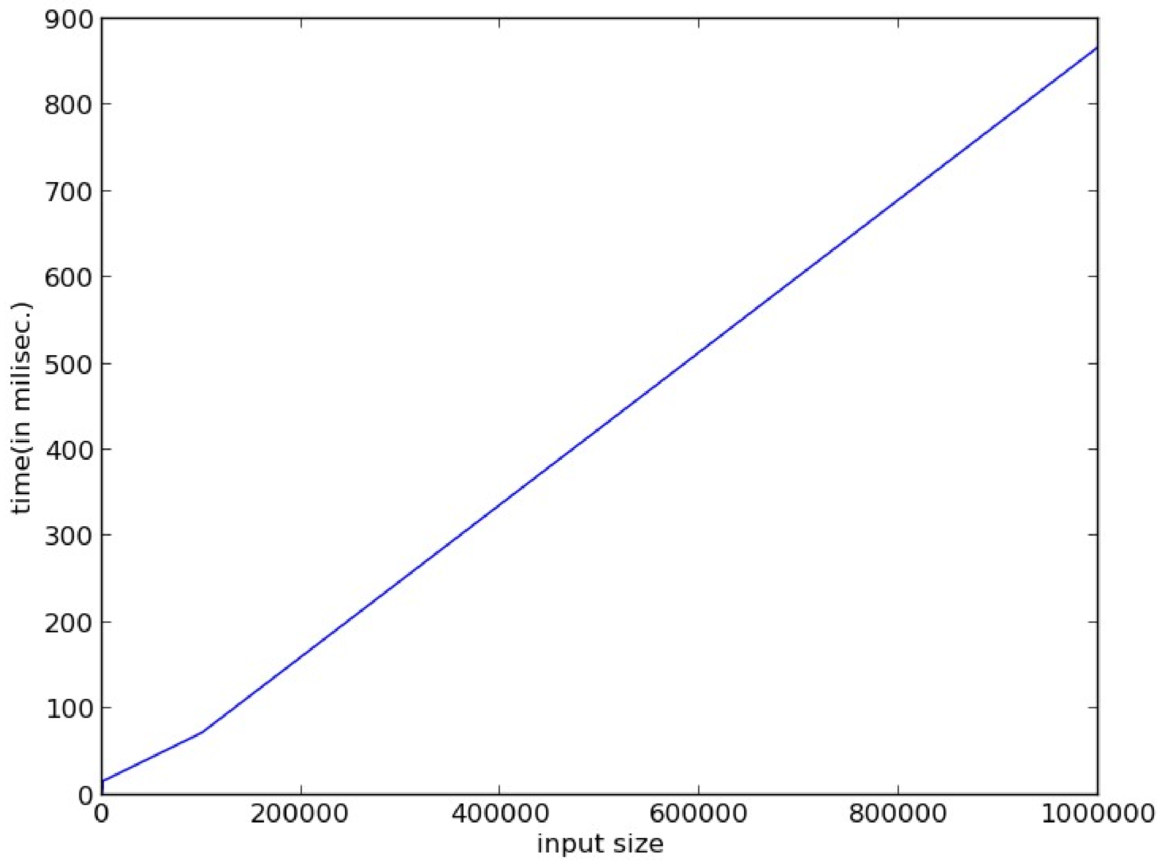
1. AVL for testgen :-



2. **AVL for testgenIS:**



3. B + Tree simple insertion:


input size
time(in milisec.)