CS261P PROJECT 2 REPORT

TREE ALGORITHMS

Contents:

THEORETICAL ANALYSIS

I. Binary Search Tree

Binary Search Tree is a node-based binary tree data structure which has the following properties:

- The left subtree of a node contains only nodes with keys lesser than the node's key.
- The right subtree of a node contains only nodes with keys greater than the node's key.
- The left and right subtree each must also be a binary search tree.
- There must be no duplicate nodes.

Pseudo Code:

```
def add(root, val):
        if (root == null) {
                return new TreeNode<T>(val);
        }
        int cmp = val.compareTo(root.value);
        if (cmp < 0)
           root.left        = add(root.left,          val);
        else if (cmp > 0)
           root.right = add(root.right, val);
        else
           root.value        = val;
        root.size = 1 + size(root.left) + size(root.right); return root;

def remove(root, key):
      if (x == null) return null;
      int cmp = key.compareTo(x.value);
      if (cmp < 0) x.left = remove(x.left, key); else if (cmp > 0)
      x.right = remove(x.right, key); else {
        if (x.right == null) return x.left;
            if (x.left          == null) return x.right;
```

```
            TreeNode<T> t = x;
            x = min(t.right);
            x.right = deleteMin(t.right);
            x.left = (t.left);
        }
        x.size = size(x.left) + size(x.right) + 1;
        return x;

def search(k):
          low = 0
          high = n-1
          while low <= high
                    mid = (low + high)/2
                    // compare k with A[mid]:
                    if k < A[mid]: high = mid - 1
                    else if k > A[mid]: low = mid + 1
                    else: return mid
k is not in data
```

Algorithm Analysis:

Binary Search Tree - Tree Insertions in Random Order

- Expected Cost Assume keys are 0,1,. . . ,n – 1.
- When we insert key x . . . The expected cost of the insertion is E [the number of nodes encountered on the search path for x]

- To compute the expected number of nodes encountered on the search path for x:

For any other key y, let C(y) = 1 if we encounter y when searching for x, C(y) = 0 otherwise.

So average insertion cost is O log n) if insertions are in random order and there are no deletions.

II. AVL Trees

In an AVL Tree, at each node, the height of the left subtree and the height of the right subtree differ by at most one. Each node stores either

- Its height;
- Whether its height differs from the height of its parent by 1 or 2 (A single bit of information)

Number of nodes in AVL tree ≥ Fib (h) – 1.

This implementation of AVL is implemented using templates, therefore supports any type of values to be passed.

Pseudo Code:

```
def add(root, val):
        if (x == null) return new TreeNode<T>(val, 0, 1); int cmp
        = val.compareTo(x.value); if (cmp < 0) {

                x.left = put(x.left, val);
        }
        else if (cmp > 0) {
                x.right = put(x.right, val);
        }
        else {
                x.value = val;
                return x;
        }
        x.size = 1 + size(x.left) + size(x.right);
        x.height = 1 + Math.max(height(x.left), height(x.right)); return
        balance(x);

def balance(x):
        if (balanceFactor(x) < -1) {
                if (balanceFactor(x.right) > 0) {
                        x.right = rotateRight(x.right);
                }
                x = rotateLeft(x);
        }
```

```
        else if (balanceFactor(x) > 1) {
                if (balanceFactor(x.left) < 0) {
                    x.left = rotateLeft(x.left);
            }
            x = rotateRight(x);
        }
        return x;

def balanceFactor(x):
        return height(x.left) - height(x.right);

def remove(root, key):
        int cmp = key.compareTo(x.value);
        if (cmp < 0) {
            x.left = remove(x.left, key);
        }
        else if (cmp > 0) {
            x.right = remove(x.right, key);
        }
        else {
            if (x.left == null) {
                return x.right;
            }
            else if (x.right == null) {
                return x.left;
            }
            else {
                TreeNode<T>y = x;
                x = min(y.right);
                x.right = deleteMin(y.right);
                x.left = y.left;
            }
        }
        x.size = 1 + size(x.left) + size(x.right);
        x.height = 1 + Math.max(height(x.left), height(x.right)); return
        balance(x);


def deleteMin(x)
        if (x.left == null) return x.right;
        x.left = deleteMin(x.left);
        x.size = 1 + size(x.left) + size(x.right);
```

```
    x.height = 1 + Math.max(height(x.left), height(x.right)); return
    balance(x);


def search(root, k):
        if (x == null) return null;
        int cmp = key.compareTo(x.value);
        if (cmp < 0) return get(x.left, key);
        else if (cmp > 0) return get(x.right, key);
      else return x;
```

Algorithm Analysis:

To insert into an AVL Tree:

- Do standard insertion.
- Node gets inserted at leaf.
- Follow a path from the leaf back up to the root, updating heights
- If a node has become unbalanced, do a rotation to rebalance it.
- Case analysis shows: never need more than two rotations.

Deletion

- Follows the same basic principle but is messier.

May need to do as many as O (log n) rotations.

III. Splay Trees

- No explicit balance condition
- Amortized (not worst-case) time bound on operations

- Basic operation is splay. Splaying at node x means making x the root through a specific series of rotations.

- Note: we cannot use an arbitrary sequence of rotations that brings x to the root. The analysis depends on using the specific sequence described here.

- 3 cases:

1. x has no grandparent (zig)
2. x is LL or RR grandchild (zig-zig)
3. x is LR or RL grandchild (zig-zag)

Splay tree operations:

like binary search tree, except . . .

- After access(i):

• Successful: splay at node containing i

• Unsuccessful: splay at last node on search path

- After insert(i): splay at node containing i

- After delete(i):

• Found: Let x be the node that was actually deleted; splay at the node that was the parent of x

• Not found: Splay at last node on search path

- As we will show next, each of these operations can be performed in O(log n) amortized time.

- NOTE: Nice property of splay trees: can make any given node the root (at cost of O(log n) amortized time).

This implementation of Splay Tree is implemented using templates**,** therefore supports any type of values to be passed.

Pseudo Code:

```
def add(root, val):

        TreeNode<T> n;
        int c;
        if (root == null) {
            root = new TreeNode<T>(key);
    return;

def remove(key):
        TreeNode<T> x;
        splay(key);
        if (key.compareTo(root.value) != o) {
            return;
        }
        ||   Now delete the root if
        (root.left == null) {
            root = root.right; }
        else {
            x  = root.right;
            root = root.left;
            splay(key);
            root.right = x;
        }

def search(root, key):
    if (root == null) return null;
    splay(key);
    if(root.value.compareTo(key) != o) return null; return root;
```

Algorithm Analysis:

For a node w in a splay tree, let |w| be the number of external nodes descending from w (i.e., one more than the number of keys in the subtree rooted at w).

The change in potential for an operation is the sum of:

1. The change in potential prior to the splay (due to the basic binary tree operation)
2. The change in potential during the splay

We will analyze these two terms separately.

Amortized analysis:

ΔΦ prior to the splay.

How does the potential change when we do a search, delete, or insert, prior to the splay?

- Search: no change

Delete: decreases

ΔΦ during the splay.

Lemma: Let ΔΦ be the change in the potential function due to one of the rebalance operations shown for Case 1, Case 2, or Case 3, and let r be the factor by which |x| increases during that operation. Then:

- In Case 1,
  $\Delta\Phi - \lg r^3 \leq 0$
- In Cases 2 and 3,
  $\Delta\Phi - \lg r^3 \leq -2$

Corollary: Let d be the initial distance from x to the root. Then when we splay at x, so that it becomes the root, the change in the potential function is

$\Delta\Phi \leq -(d - 1) + 3 \lg(n + 1)$.

Theorem: The amortized cost of a search, insert, or delete operation on a splay tree is O(log n), where n is the number of elements present.

Proof:

- Actual time: O(d) (d =distance from splayed node to root node, immediately before the splay).
- Change in potential:
    - Before the splay:
        - Search: no change
        - Insert: +O(log n)
        - Delete: decrease
    - During the splay: $\Delta\Phi \leq -d + 1 + 3 \lg(n + 1)$

Hence amortized time is O(log n).

IV. Treap

This is another variation of a binary search tree in which the key is arranged according to its value. But the balancing is done using the priority values associated with the nodes and a max heap is maintained such that each parent node has higher priority than its child.

In a Treap the balancing is done on the priority values and the parent should always have higher priority. So now after an insert or a deletion if that property is not maintained then again we balance the tree using right or left rotations but based on the priority values this time and maintaining a max heap.

Each node has a key value associated with it and a priority value indicating its priority, a left child and a right child.

Insert(key)

The key value is inserted in a similar fashion to binary search tree by maintaining the property that the left value should be less than root and the right should be greater than the root. After that the priority is checked and a max heap should be followed which means that the root priority has to be greater in value than the children. If that is violated, then we balance the tree using left and right rotations on the priority value.

```
if(node==null)
        return new TreeNode(key);
if(node.key>key)
{
        node.left=insert(node.left,key);
        if(node.priority<node.left.priority)
        {
                node=RightRotate(node);
        }
}
else
{
        node.right=insert(node.right,key);
        if(node.priority<node.right.priority)
        {
                node=LeftRotate(node);
        }
}
return node;
```

Search(key)

This is quite similar to the search used for binary search tree. The node value is compared to the key and recursively is compared to the left or right node depending if the key is greater or less than the node value. Then if found return true else return false.

```
if(node == null)
        return false;
else if(node.key == key)
        return true;
```

```
else if(key < node.key)
        return search(node.left,key);
else
        return search(node.right,key);
```

Delete(key)

The key to be deleted is first found. Then there will be three cases depending on whether the node is a leaf, has one child or has two children. So it has three cases: -

1) When it is a leaf: In this case just delete the node.

2) When it has one child: In this case we make the child as the root by making left or right rotation depending on whichever child exists and then delete the node after making it the leaf.

3) When it has two children: In this case we make the child with the higher priority as the root by using left and right rotations and then delete the node.

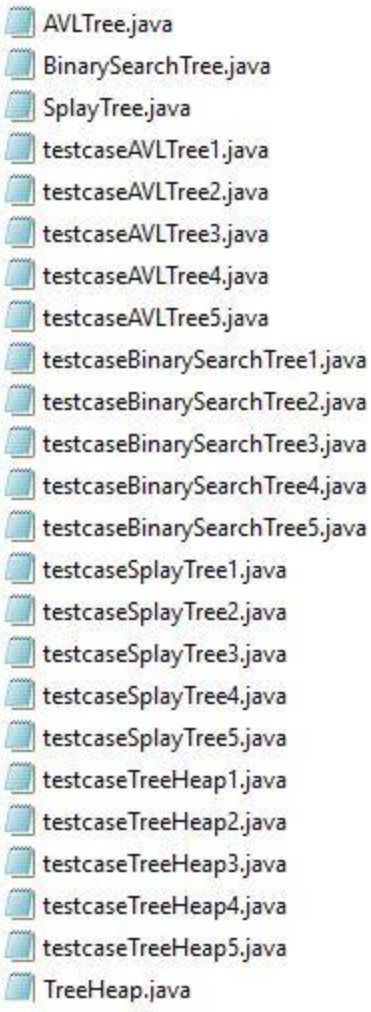
```
if(root==null)
        return root;
if(key<root.key)
        root.left=delete(root.left,key);
else if(key>root.key)
        root.right=delete(root.right,key);
else
{
        if(root.right!=null || root.left!=null)
        {
                if(root.left==null)
        {
                root=LeftRotate(root);
                root.left=delete(root.left,key);
        }
        else if(root.right==null || (root.left.priority>root.right.priority))
        {
                root=RightRotate(root);
                root.right=delete(root.right,key);
        }
        else
        {
                root=LeftRotate(root);
```

```
                root.left=delete(root.left,key);
        }
}
        else
        {
                return null;
        }
}

return root;
```

EXPERIMENTAL ANALYSIS

I. Running the test cases



The 'TreeOperations.java' file is an abstract class which is implemented by all the hash function files

'BinarySearchTree.java' implements Binary Search Tree

'AVLTree.java' implements AVL Tree

'SplayTree.java' implements Splay Tree

'TreeHeap.java' implements a Treap

There are 5 test cases for each algorithm

The test cases are explained below

There is also a Makefile included to build the program
run the 'make' command on terminal to build the program

Any of the test case files can be run on the terminal using the command

java <testcasename>

example: *java testcaseBinarySearchTree1* will run the first test case of Binary Search Tree

The first test case of each algorithm checks for the correctness of the algorithm

The second test case implements the algorithms for 100 operations of each kind of operations insert(), search() and delete() .

The second test case implements the algorithms for 1000 operations of each kind of operations insert(), search() and delete() .

The second test case implements the algorithms for 10000 operations of each kind of operations insert(), search() and delete() .

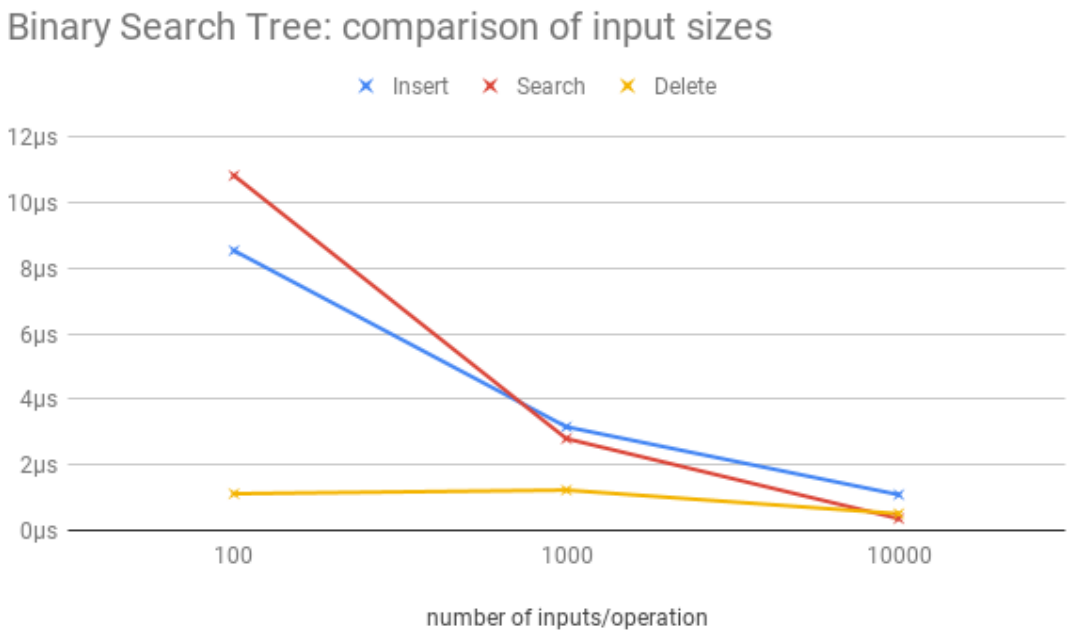
The fifth test case of all algorithms calculates the worst-case time for each operation

II. Visualization of Analysis

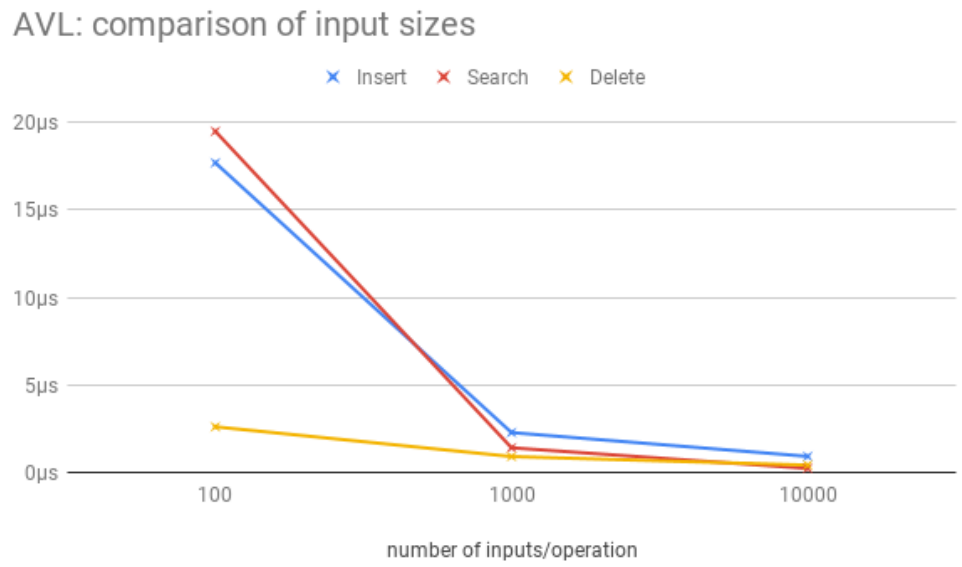
Several graphs have been plotted to compare the input size to the execution time per operation

We can observe some interesting trends

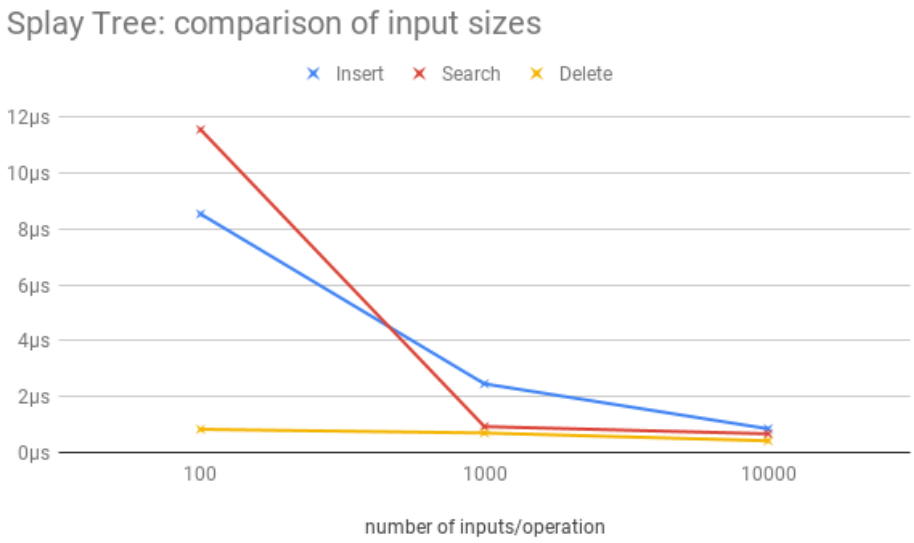
The graphs are presented below



We can see that the delete() operation is the best

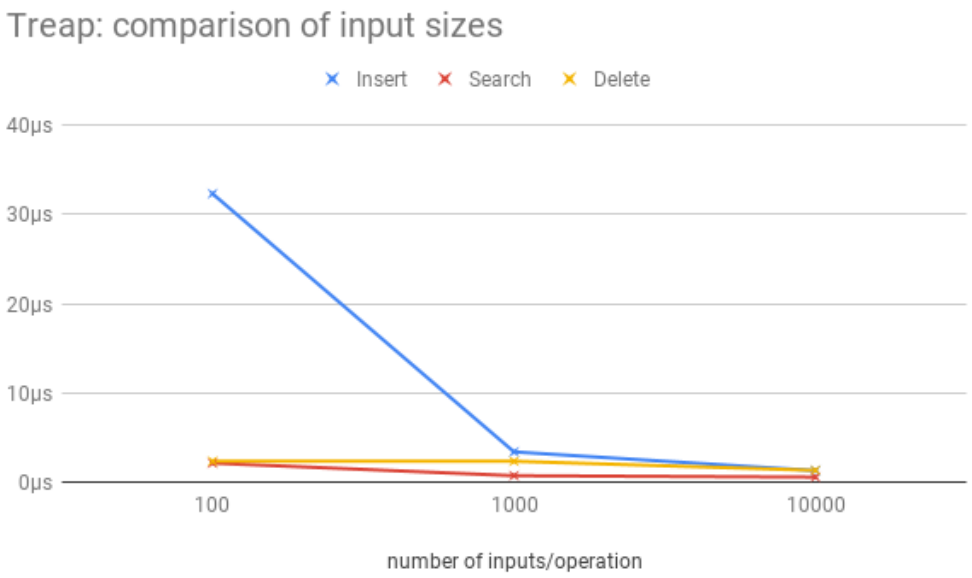


We can see that the delete() operation is the best

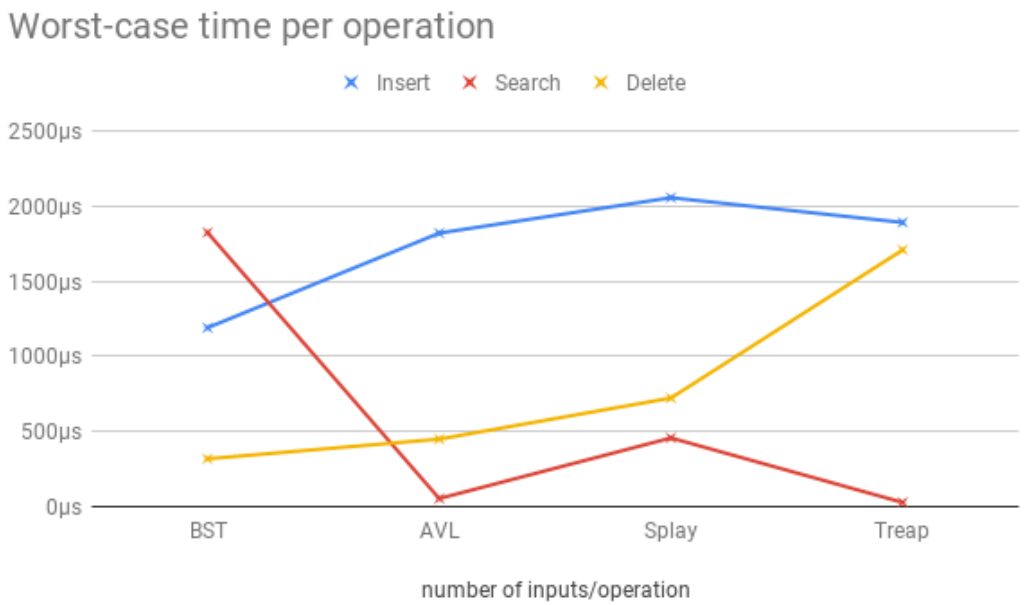


We can see that the delete() operation is the best

It is almost linear.



We can see that the delete() operation is the best



This graph compares the worst-case time per operation of insert(), search() and delete() for all the tree algorithms.

We can see that Splay Tree proves to give the lowest times compared to all the other approaches. This is the case because the size of input was 1000. Therefore at small input sizes the Splay Tree works well, but for larger input sizes it will be slower.

III. General Conclusions

- All the tree algorithms have an Amortized time of O (log n) for all the operations, but in certain specific cases some algorithms perform better than others
- Splay Trees are good approach when the input size is small

In general trees are a great way to store data if you want a decent amortized time. Searching is very fast and therefore hash tables are widely used in industry applications.