261Proj2

Pseudo Code

Binary Search Tree

```
public class BinaryTree {
    private Node head;
    public BinaryTree(){
        set default values
    }
    public Node search(int v){
        Node nodeV = head;
        while(nodeV!=null){
            if(nodeV.getV()<v) nodeV=nodeV.getRightChi();
            else if(nodeV.getV()>v)nodeV=nodeV.getLeftChi();
            else return nodeV;
        }
        return null;
    }
    public Node insert(int v){
        Node nodeV = head;
        if(nodeV==null){
            return new Node(v);
        }
        while(true){
            // search for the correct direction (left or right according to comparison
between val and nodeV.value ), insert when find empty position
            if(nodeV.getV()<v){
                if(nodeV.getRightChi()==null){
                    nodeV.setRightChi(new Node(v, nodeV));
                    return nodeV.getRightChi();
                }
                else nodeV=nodeV.getRightChi();
            }
            else if(nodeV.getV()>v){
                if(nodeV.getLeftChi()==null){
                    nodeV.setLeftChi(new Node(v, nodeV));
                    return nodeV.getLeftChi();
                }
                else nodeV=nodeV.getLeftChi();
            }
            else return null;
        }
    }
    public boolean delete(int v){
        // serach for v
        Node nodeV = search(v);
        // if v doesn't exist return false
        if(nodeV==null) return false;
        if(node v doesn't have left childl){
            let v's right child replace v
        }else if(node v doesn't have right childl){
            let v's left child replace v
        }else (then v has both left and right child){
            find the most left element in the tree with root v.rightChi and replace v.
        }
    }
}
```

AVL Tree

```
public class AVLTree{
    private
    private AVLNode root;
    //recursively search for val in the tree with root rt
    private AVLNode searchRcur(int val, AVLNode rt){
        if(rt==null) return null;
        if(val<rt.val) return searchRcur(val, rt.left);
        else if(val>rt.val) return searchRcur(val, rt.right);
        else return rt;
    }
    private int height(AVLNode rt){
        if(rt==null) return 0;
        else return rt.height;
    }
    //when the unbalance is rt's left child's left child height higher (LL)
    //just rotate right at r's leftChi
    //and update the height
    private AVLNode rotLL(AVLNode rt){
        rotCallCnt++;
        AVLNode rt1 = rt.left;
        rt.left = rt1.right;
        rt1.right = rt;
        rt1.height = Math.max(height(rt1.left), height(rt1.right))+1;
        rt.height = Math.max(height(rt.left), height(rt.right))+1;
        return rt1;
    }
    //when the unbalance is rt's right child's right child height higher (RR)
    //just rotate left at r's rightChi
    //and update the height
    private AVLNode rotRR(AVLNode rt){
        rotCallCnt++;
        AVLNode rt1 = rt.right;
        rt.right = rt1.left;
        rt1.left = rt;
        rt1.height = Math.max(height(rt1.left), height(rt1.right))+1;
        rt.height = Math.max(height(rt.left), height(rt.right))+1;
        return rt1;
    }

    //when the unbalance is rt's left child's right child height higher (LR)
    //just rotRR at rt's leftChi and then rotLL at rt
    private AVLNode rotLR(AVLNode rt){
        rt.left = rotRR(rt.left);
        return rotLL(rt);
    }
    //when the unbalance is rt's right child's left child height higher (LR)
    //just rotLL at rt's rightChi and then rotRR at rt
    private AVLNode rotRL(AVLNode rt){
        rt.right = rotLL(rt.right);
        return rotRR(rt);
    }

    private AVLNode insertRcur(int val, AVLNode rt){

        // recursively insert val in the right subtree.
        // check whether the current node's (rt's) left and right is balanced or not.
        insertRecurCallCnt++;
        if(rt==null) return new AVLNode(val, null, null);
        if(val<rt.val) {
            rt.left = insertRcur(val, rt.left);
            if (height(rt.left) - height(rt.right) == 2) {
                if (val < rt.left.val) {

                // if inserted node is in rt's left's left subtree and rt's left is higher 2
```

```
than right: rotLL(rt)
                    rt = rotLL(rt);
                } else {

                // if inserted node is in rt's left's right subtree and rt's left is higher
2 than right: rotLR(rt)
                    rt = rotLR(rt);
                }
            }
        }
        else if(val>rt.val){
            rt.right = insertRcur(val, rt.right);
            if(height(rt.right)-height(rt.left)==2){
                if(val > rt.right.val){

                    // if inserted node is in rt's right's right subtree and rt's right is
higher 2 than left: rotRR(rt)
                    rt = rotRR(rt);
                }
                else{

                    // if inserted node is in rt's right's left subtree and rt's right is
higher 2 than left: rotRL(rt)
                    rt = rotRL(rt);
                }
            }
        }

        // update rt's height
        rt.height = Math.max(height(rt.left), height(rt.right))+1;
        return rt;
    }

    //search for the rightest ele
    public AVLNode searchMax(AVLNode rt){
        AVLNode tmpr = rt;
        while(tmpr.right!=null){
            searchMinMaxIterCnt++;
            tmpr = tmpr.right;
        }
        return tmpr;
    }
    //search for the leftest ele
    public AVLNode searchMin(AVLNode rt){
        AVLNode tmpr = rt;
        while(tmpr.left!=null){
            searchMinMaxIterCnt++;
            tmpr = tmpr.left;
        }
        return tmpr;
    }
    // Very similar to the Insert(). Recursively find the ele to be deleted. Check whether
unbalance exists at the current node. According to the LL,LR,RR,RL cases do the rotation
calls. Update the height.
    private AVLNode deleteRcur(int val, AVLNode rt){
        deleteRecurCallCnt++;
        if(rt==null) return null;
        if(val>rt.val) {
            rt.right = deleteRcur(val, rt.right);
            if (height(rt.left) - height(rt.right) == 2) {
                AVLNode tmpr = rt.left;
                if (height(tmpr.left) > height(tmpr.right)) {
                    rt = rotLL(rt);
                } else {
                    rt = rotLR(rt);
                }
            }
        }
```

```
        else if(val<rt.val){
            rt.left = deleteRcur(val, rt.left);
            if(height(rt.right) - height(rt.left)==2){
                AVLNode tmpr = rt.right;
                if(height(tmpr.left) < height(tmpr.right)){
                    rt = rotRR(rt);
                }
                else{
                    rt = rotRL(rt);
                }
            }
        }
        // the current node to be deleted
        else{
            // if the node to be deleted have both left and right child, we replace it with
another val (leftChi's max or rightChi's min, according to which side is higher. Such choice
would not lead the tree to be unbalanced) and then delete that val in the subtree.
            if((rt.left!=null) && (rt.right!=null)) {
                // if left subtree higher than right subtree, then find the leftMax as the
new rt
                // and then recursively delete leftMax (who has at least one null-child)
                if (height(rt.left) > height(rt.right)) {
                    AVLNode tmprt = searchMax(rt.left);
                    rt.val = tmprt.val;
                    rt.left = deleteRcur(tmprt.val, rt.left);
                } else {
                // if right subtree highger than left subtree, then find the rightMin as the
new rt
                // and then recursively delete rightMin (who has at least one null-child)
                    AVLNode tmprt = searchMin(rt.right);
                    rt.val = tmprt.val;
                    rt.right = deleteRcur(tmprt.val, rt.right);
                }
            }
            // if the node to be deleted has null child, just replace the node with another
child.
            else{
                if(rt.left==null){
                    rt = rt.right;
                }
                else{
                    rt = rt.left;
                }
            }
        }
        // update the height
        if(rt!=null) rt.height = Math.max(height(rt.left), height(rt.right));
        return rt;
    }
}
```

Splay Tree

```
//This implement is based on Top-Down Splay Operation and need to maitain 3 part trees
during Splay().
public class SplayTree {
    private void splay(int val){
        root = splay(root, val);
    }
    // Left Rotation at grandp and split the parent with the midTree
    private void rotLChi(SplNode grandp, SplNode parent){
        grandp.left = parent.right;
        parent.right = grandp;
        //split the parent with middle tree
        parent.left = null;
```

```
    }
    // Right Rotation at grandp and split the partent with midTree.
    private void rotRChi(SplNode grandp, SplNode parent){
        grandp.right = parent.left;
        parent.left = grandp;
        //split the parent with middle tree
        parent.right = null;
    }
    // top-down splay val in the tree with root rt
    private SplNode splay(SplNode rt, int val) {
        if (rt == null) return null;
        SplNode pseuNode = new SplNode(Integer.MAX_VALUE);
        // 3 part trees: a tree of all ele smaller than current node, a tree of all ele
larger than current node, and a middle part with current node is the root.
        // also maintain the rightest (max) node, the leftest (min) node, and the current
node for each tree seperately, just for the final combination.
        SplNode leftMax = pseuNode;
        SplNode rightMin = pseuNode;
        SplNode t = rt;


        while (true) {
            splayIterCallCnt++;
            if (val == t.val) {
                break;
            }
            else if (val < t.val) {
                // if target node is in its grandp's left subtree
                //Note: the variable parent is target's parent, the variable t is target's
grandp
                SplNode parent = t.left;
                if (parent == null) {
                    break;
                } else {
                    if (val < parent.val) {
                    // if target should be in its granpdp's left's left subtree
                        if (parent.left == null) {
                            //if no target node found, splay its parent up.
                            //zig, do single rotation
                            t.left = null;
                            rightMin.left = t;
                            rightMin = t;
                            t = parent;
                        } else {
                            // target is in grandp's left's left subtree
                            // zig-zig
                            SplNode tmp = parent.left;
                            rotLChi(t, parent);
                            //update right tree and its min node
                            rightMin.left = parent;
                            rightMin = parent;
                            //update the middle tree's root
                            t = tmp;
                        }
                    } else {
                    //target should be in grandp's left's right subtree
                        //zig or zig-zag(simplified to zig)
                        t.left = null;
                        rightMin.left = t;
                        rightMin = t;
                        t = parent;
                    }
                }
            }
            else {
                // if target node is in its grandp's right subtree
                // Very similar to the counterpart above
                //val>t.val
```

```
                SplNode parent = t.right;
                if (parent == null) {
                    break;
                } else {
                    if (val > parent.val) {
                        if (parent.right == null) {
                            //zag
                            t.right = null;
                            leftMax.right = t;
                            leftMax = t;
                            t = parent;
                        } else {
                            //zag-zag
                            SplNode tmp = parent.right;
                            rotRChi(t, parent);
                            //update left tree and its max node
                            leftMax.right = parent;
                            leftMax = parent;
                            //update the middle tree's root
                            t = tmp;
                        }
                    } else {
                        //val <= t.right.val
                        //zag or zag-zig (simplified to zag)
                        t.right = null;
                        leftMax.right = t;
                        leftMax = t;
                        t = parent;
                    }
                }
            }
        }
        //re-assemble / combination of 3 parts
        leftMax.right = t.left;
        rightMin.left = t.right;
        t.left = pseuNode.right;
        t.right = pseuNode.left;
        return t;
    }

    //search val and do splay
    public SplNode search(int val){
        if(root==null) return null;
        splay(val);
        return root;
    }

    //insert val and do splay
    public void insert(int val){
        root = insertRecur(val, root);
        splay(val);
    }
    private SplNode insertRecur(int val, SplNode rt){
        if(rt==null ) return new SplNode(val, null, null);
        while(true){
            insertRecurCallCnt++;
            if(rt.val==val) break;
            else if(rt.val<val){
                if(rt.right==null){
                    rt.right = new SplNode(val, null, null);
                    break;
                }
                else rt=rt.right;
            }
            else{
                if(rt.left==null){
                    rt.left = new SplNode(val, null, null);
                    break;
```

```
                }
                else rt=rt.left;
            }
        }

        return root;
    }
    // delete val and do splay
    public void delete(int val){
        root = deleteNormal(root, val);
        splay(val);
    }
    private SplNode searchMin(SplNode rt){
        if(rt==null) return null;
        while(rt.left!=null) rt=rt.left;
        return rt;
    }
    // just delete w/o splay
    private SplNode deleteNormal(SplNode rt, int val){
        deleteRecurCallCnt++;
        if(rt==null) return null;
        if(rt.val==val) {
            if (rt.left == null) return rt.right;
            if (rt.right == null) return rt.left;
            SplNode tmprt = searchMin(rt.right);
            rt.val = tmprt.val;
            rt.right = deleteNormal(rt.right, tmprt.val);
            return rt;
        }
        else if(rt.val < val){
            rt.right = deleteNormal(rt.right, val);
            return rt;
        }
        else{
            rt.left = deleteNormal(rt.left, val);
            return rt;
        }
    }
    public void inorderPrintTree(){
        inorderTrav(root);
        System.out.println();
    }
    private void inorderTrav(SplNode root){
        SplNode tmpP = root;
        if(tmpP!=null){
            inorderTrav(tmpP.left);
            System.out.print(tmpP.val+" ");
            inorderTrav(tmpP.right);
        }
    }
}
```

Treap

```
public class Treap {
    private TrpNode root;
    //right rotation at rt
    private TrpNode rotR(TrpNode rt){
        rotIterCallCnt++;
        TrpNode rt1 = rt.left;
        rt1.parent = rt.parent;
        if(rt1.parent==null) root=rt1;
        else if(rt.parent.left==rt) rt.parent.setLChi(rt1);
        else rt.parent.setRChi(rt1);
        rt.setLChi(rt1.right);
```

```
        rt1.setRChi(rt);
        return rt1;
    }
    //left rotation at rt
    private TrpNode rotL(TrpNode rt){
        rotIterCallCnt++;
        TrpNode rt1 = rt.right;
        rt1.parent = rt.parent;
        if(rt1.parent==null) root=rt1;
        else if(rt.parent.left==rt) rt.parent.setLChi(rt1);
        else rt.parent.setRChi(rt1);
        rt.setRChi(rt1.left);
        rt1.setLChi(rt);
        return rt1;
    }
    //search for val iteratively
    public TrpNode search(int val){
        TrpNode rt = root;
        while(rt!=null && rt.val!=val){
            searchRcurCallCnt++;
            if(val<rt.val) rt=rt.left;
            else rt=rt.right;
        }
        if(rt==null) return null;
        else return rt;
    }
    // insert val iteratively
    public void insert(int val){
        if(root==null){
            root = new TrpNode(val);
            return;
        }
        TrpNode rt = root;
        while(true){
            insertRecurCallCnt++;
            if(val<rt.val){
                if(rt.left==null){
                    rt.setLChi(val);
                    rt = rt.left;
                    break;
                }
                else rt=rt.left;
            }
            else if(val>rt.val){
                if(rt.right==null){
                    rt.setRChi(val);
                    rt = rt.right;
                    break;
                }
                else rt=rt.right;
            }
            else return;
        }

        TrpNode rtpr = rt.parent;
        //from current isnerted node start to rotate if the prio-relation not satisfy the
heap requirement
        while(rtpr!=null && rt.prio > rtpr.prio){
            if(rtpr.left == rt){
                rt = rotR(rtpr);
                rtpr = rt.parent;
            }
            else{
                rt = rotL(rtpr);
                rtpr = rt.parent;
            }
        }
        return;
```

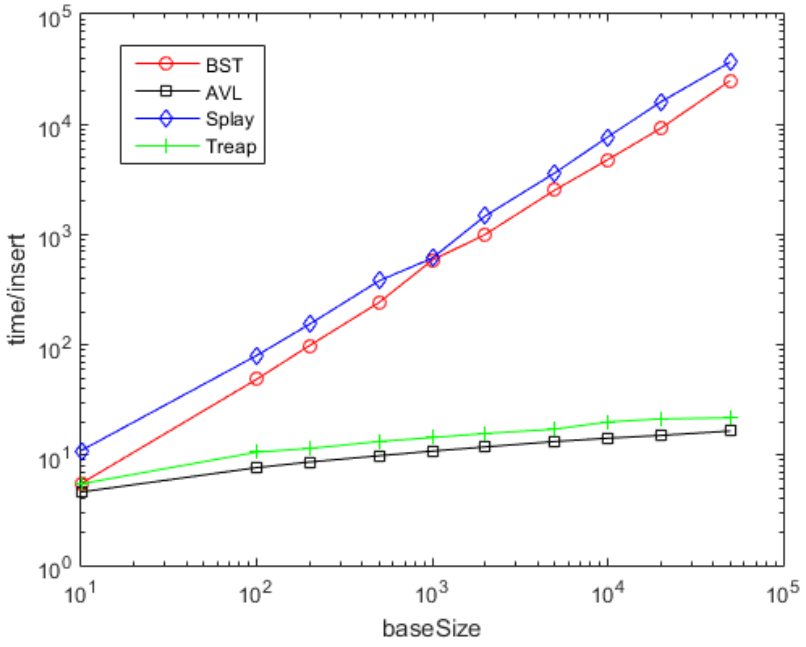
```
    }
    public void delete(int val){
        // search for val
        TrpNode rt = search(val);
        if(rt==null) return;
        // rotate nodeVal down to leafNode (just a little like minHeap SiftDown() )
        while( !(rt.left==null && rt.right==null) ){
            deleteRecurCallCnt++;
            if(rt.left==null){
                rotL(rt);
            }
            else if(rt.right==null){
                rotR(rt);
            }
            else if(rt.left.prio > rt.right.prio){
                rotR(rt);
            }
            else{
                rotL(rt);
            }
        }
        // delete rt (at leaft)
        TrpNode rtpr = rt.parent;
        if(rtpr==null) root=null;
        else if(rtpr.left==rt) rtpr.left=null;
        else rtpr.right=null;
    }
}
```

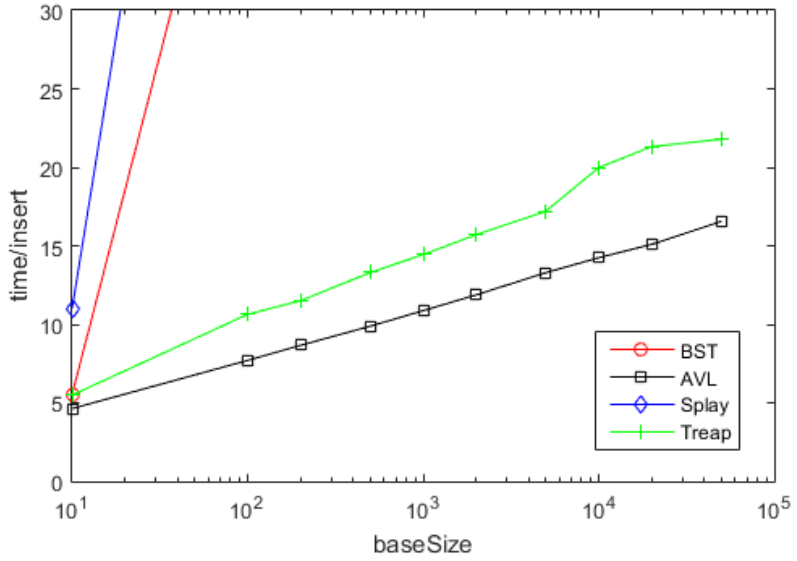
Time Cost Results and Analysis

Not that important:
1. For “TimePerOperation(insert/delete/search)”, I firstly construct a tree by isnerting a sequence (baseSize) of sorted/random-order elements, and then insert/delete/search a random element, count the basic function (such as one iteration in recursive/loop func) calls number change before and after this single operation. Repeat the process (from construct a new tree) 100 times and calculate the average of funciton calls change as the timePerOperation at this baseSize.
2. For “timeAverageAmortized”, I just isnerting a sequence of sorted/random-order elements, and save the function calls number at many points (different insertSize), and then calculate funcCallNum/insertSize as the timeAverageAmortized.
3. Sometimes a figure above is log-log and below is log-linear based on same result just for clear.

Sorted Sequence:

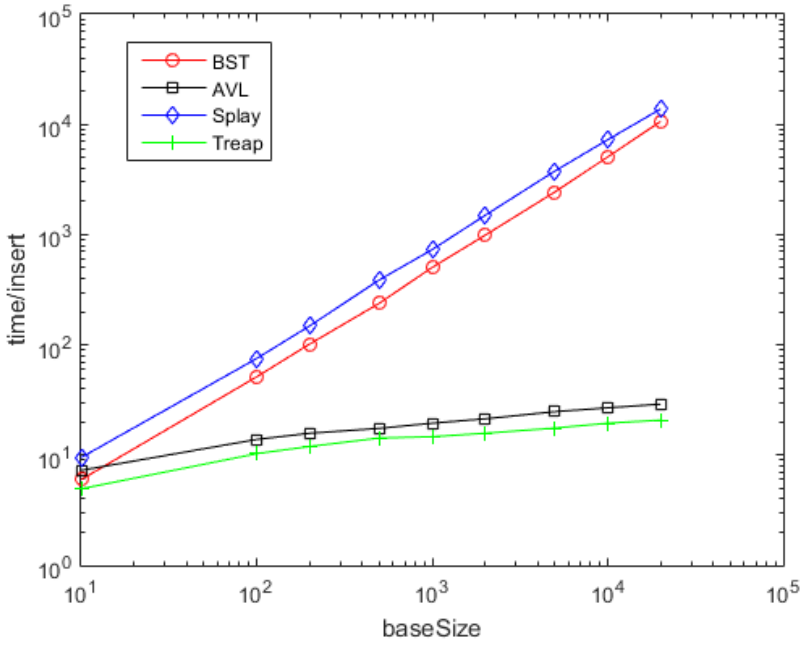
timePerInsert ~ baseSize

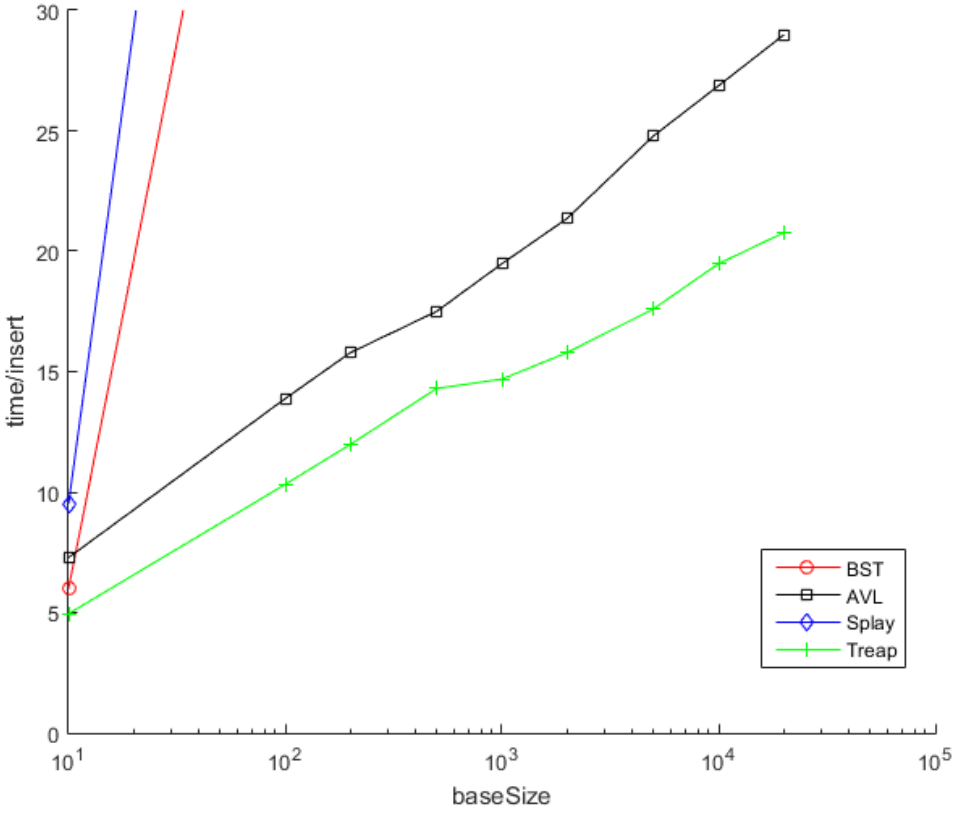




timePerInsert(BST, sorted) = O(N)
timePerInsert(Splay, sorted) = O(N)
timePerInsert(AVL, sorted) = O(logN)
timePerInsert(Treap, sorted) = O(logN)
Because BST and Splay Tree do not maintain balance, for a sorted sequence input they would both decrease to a link list. So based on this to insert another random element would take linear time. AVL Tree always keeps a strict balance while Treap keeps balance by randomized alternatives so they are both O(logN).

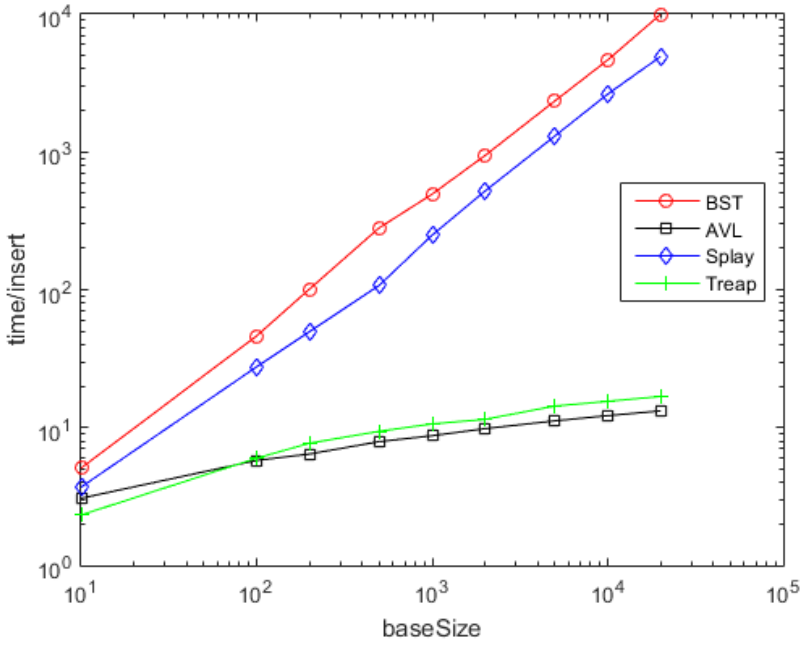
timePerDelete ~ baseSize

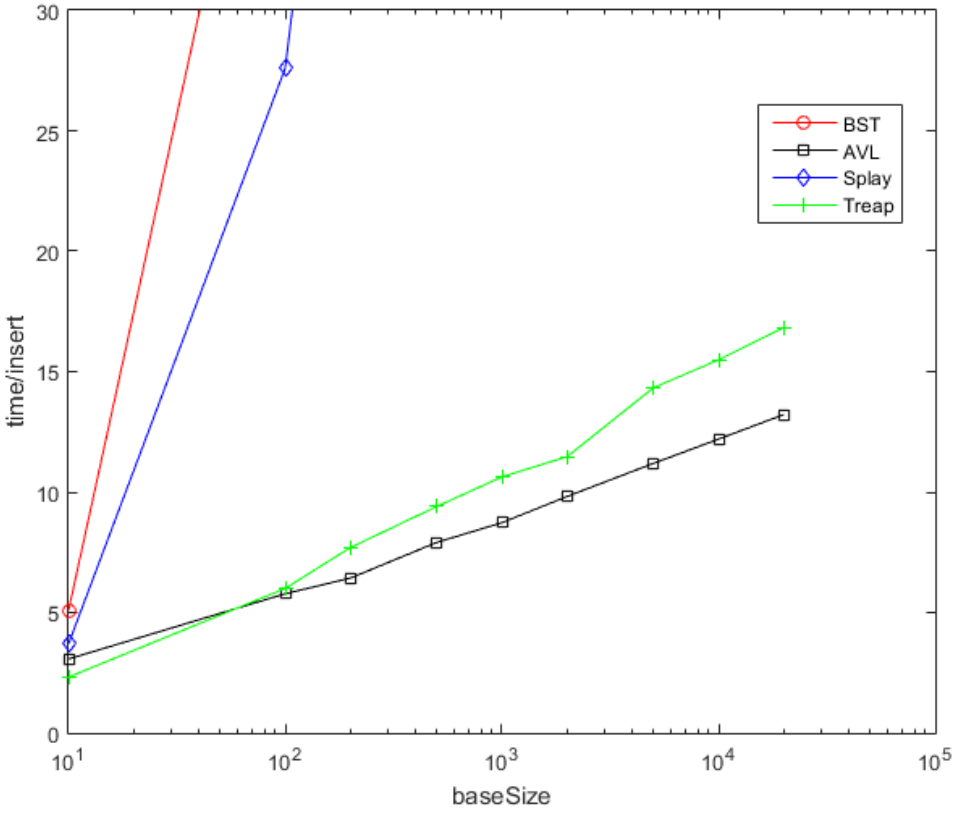




Simliar to the Insert case.

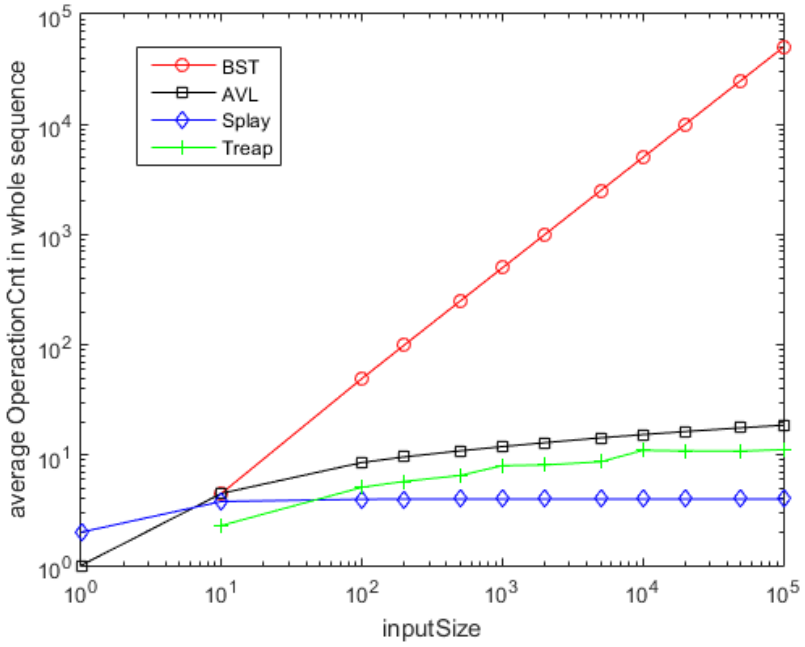
timePerSearch ~ baseSize





Simliar to the Insert case.

timeAverageAmortized ~ insertSize

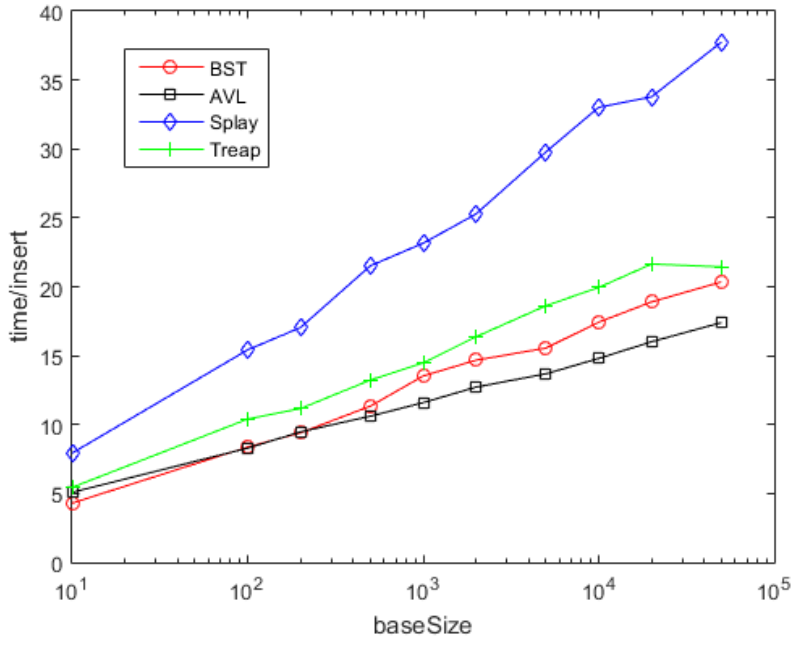


timePerInsertAmortized(BST, sorted) = O(N)
timePerInsertAmortized(Splay, sorted) = O(1)
timePerInsertAmortized(AVL, sorted) = O(logN)
timeAmortized(Treap, sorted) = O(logN)

timePerInsertAmortized(BST, sorted) = (1+2+3+...+N)/N = O(N).
For Splay Tree, every time insert a larger element, it just go to the root's right child and then splay up to be the new root. So although the tree becomes a link list, it takes only constant time for each insertion of a larger element.

Random Sequence:

timePerInsert ~ baseSize

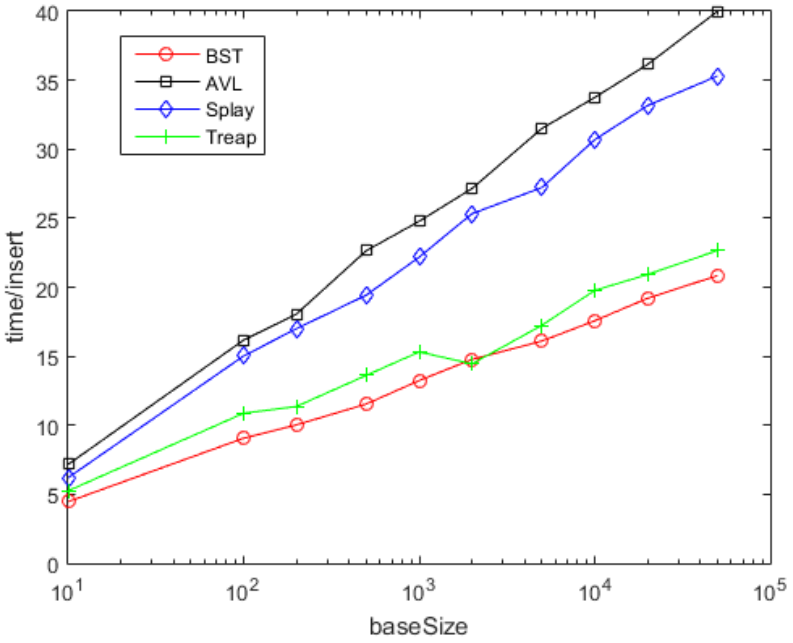


timePerInsert(BST, random) = O(logN)
timePerInsert(Splay, random) = O(logN)
timePerInsert(AVL, random) = O(logN)
timePerInsert(Treap, random) = O(logN)

For random input sequence, the trees are all balanced to some extent, time cost are all O(logN).

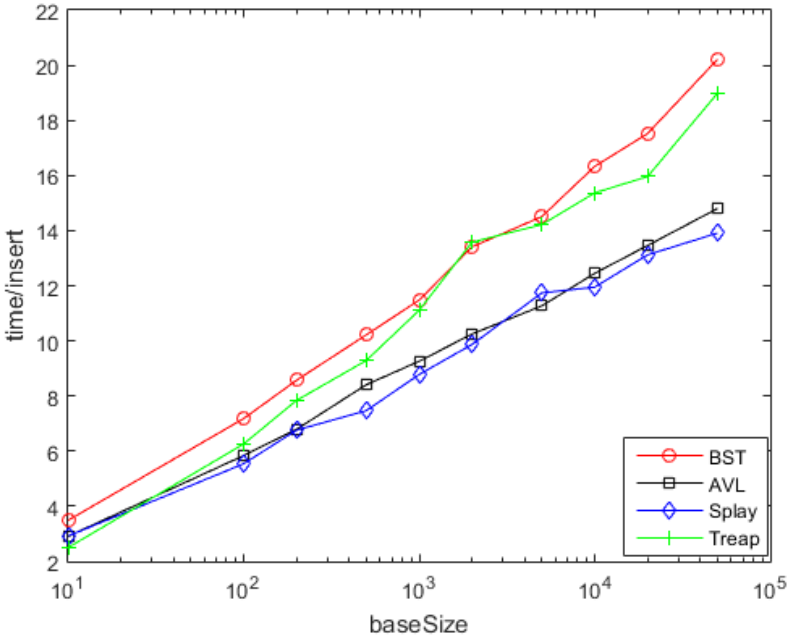
$timePerInsert(BST, random) = E[number of nodes on the search path]$
$= E(\sum_y C(y)) = \sum_y Pr[C(y) = 1]$
$= \sum_y \frac{1}{|y-x|+1}$
$<= 1 + 1/2 + 1/2 + 1/3 + 1/3 + \ldots + 1/n$
$<= 2ln(n) = O(logn)$

timePerDelete ~ baseSize



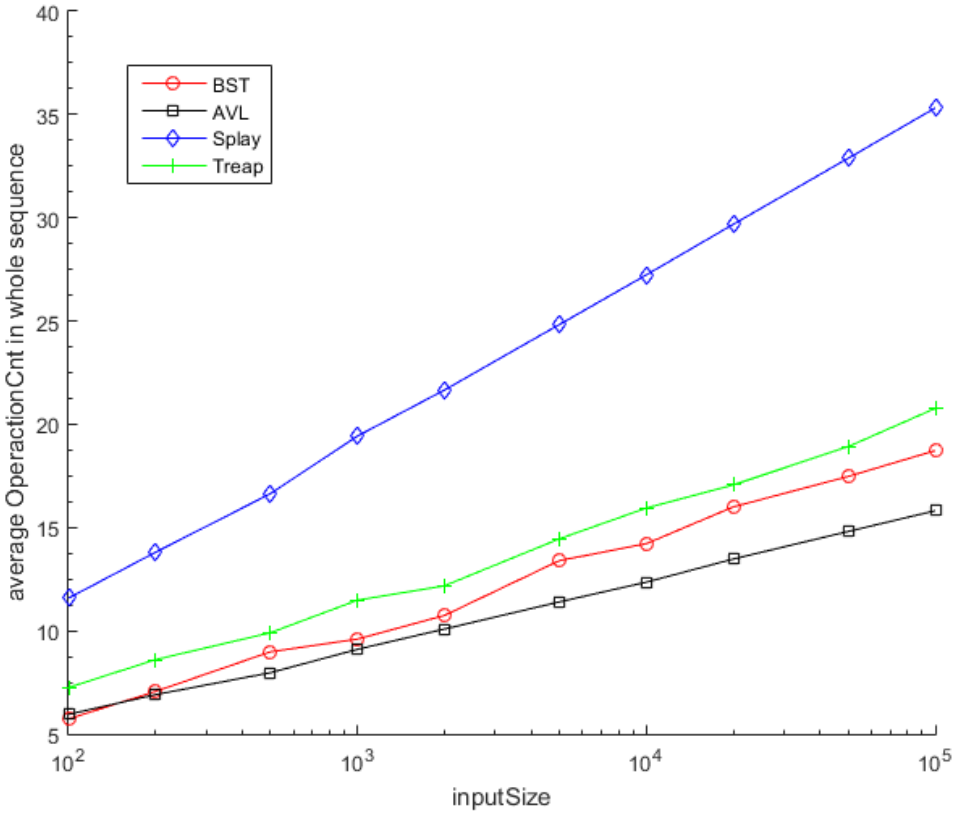
Similar to Insert case. AVL Tree cost become larger than others (while in insert case it's smaller) is because deletion may takes O(logN) rotations while insertion takes at most 2.

timePerSearch ~ baseSize



Similar to Insert case.

timeAverageAmortized ~ insertSize



timePerInsertAmortized(BST, random) = O(logN)
timePerInsertAmortized(Splay, random) = O(logN)
timePerInsertAmortized(AVL, random) = O(logN)
timeAmortized(Treap, random) = O(logN)
For Splay Tree: Let $Fai = \sum_{allnodes} lg|x|$.
We can find (1) that actual time is O(d) (d is the distance fro msplayed node to root before the splay operation)
(2) the change in potential (including 2 parts, before splay and during splay):
[a] before splay: search~0; insert~O(log n); delete~negative
[b] during splay: we can prove: $deltaFai <= -d + 1 + 3log(n+1)$
Thus the amortized time is O(log N).