AVL-based Tango tree

Yuhao Kang

Abstract

Tango tree is an efficient approach to access data in O(log log n) times. It splits data into small parts and stores each part in the auxiliary (aux) tree. Tango tree is a tree of these aux trees. In this work, we construct the tango tree based on the AVL-type aux tree and discuss relevant operations on the tango tree: cut, join, search; all of them are based on the operation of AVL tree: insert, remove, merge, split. Additionally, we will discuss the cost of each operation.

In a binary search tree (BST), the best performance of the search can be realized when the tree is balanced. For example, in the AVL tree or red-black tree, the height of the tree is O(logn). So the search operation costs O(logn). Since we do not move the recently searched node closer to the root, it will cost another O(logn) to access the node again from the root. In the AVL or RB tree, the touched node has no priority to the root.

However, In the real application, the node previously searched is more likely to be searched again. To fully make use of this property, splay tree restructures a tree so that more frequently accessed nodes are brought closer to the root.

Tango tree makes use of this property in a different way. In the following discussion, we will bring in two trees: static tree P and tango tree T.

P is the normal BST, not necessarily balanced. P does not appear in the code and is just for simplification of understanding. We do not store P.

T is the tango tree we need to manipulate, and T is a reconstructed version of P. The operation such as "cut" is complicated in T but simple in P. we will first discuss these operations in P and then generalize them to T.

The text is organized as follows: 1) how to construct T from P; 2) the information carried by nodes in T; 3) relevant operations in P and how to apply the same operations in T; 4) how to do these operations concretely in each aux tree; 5) the performance of tango tree.

I. Introduction to tango tree

Tango tree is expected to search a node in O(log log) steps. The tango tree [1] is a tree of the aux tree.

In Fig.1, we show how to construct a tango tree T from the starting point.

First, a random BST is constructed in fig.1(1). The key is {1,2,…n}. We call this tree P, the nodes are linked by the dashed line.

Then we search a series of keys {x1,x2,…xm} and highlight the path from the root to the searched key in sequence, which is defined as the preferred path. In fig.1(2), the preferred path is from root to x1. When x2 is accessed, the preferred path is altered to the path from the root to x2 in fig.1(3). The original preferred path is truncated. When the nodes are connected by these highlighted path, they form a

cluster. If nodes are connected by the dashed line, they do not belong to the same cluster. Finally, in fig.1(4), when the system go through {x1,x2,x3}, the system is split into cluster A,B1,B2…D1,D2.

Each cluster is then stored by an aux tree. There are many options for aux tree such as AVL tree, RB tree and splay tree. Here, we choose AVL tree as the form of aux tree. The dashed line connects two different aux tree. And if a node is the root of an aux tree, we mark this node.

Fig.2 shows the tango tree representation of fig.1(4). We call it tango tree T.

In P, each cluster is linear or just a point. The cluster has a height at most O(logn).

In T, the cluster was reorganized to AVL aux tree. Each aux tree has a height O(log log).

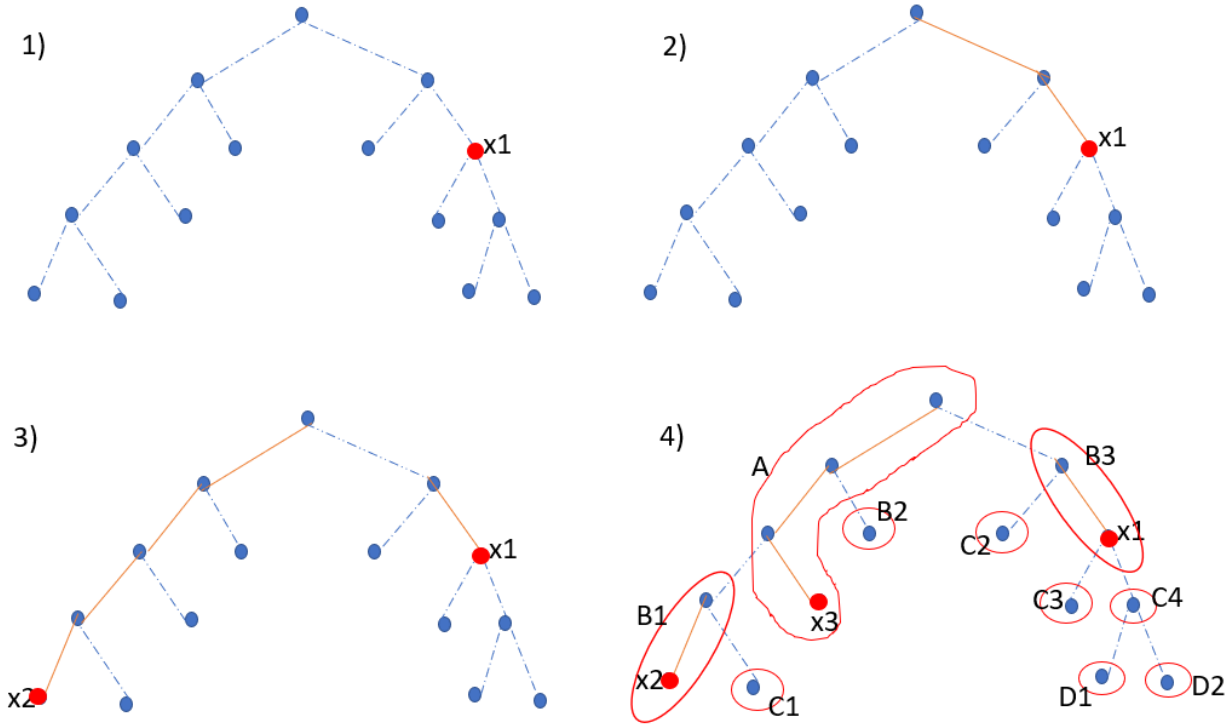


Fig.1. Static tree P.

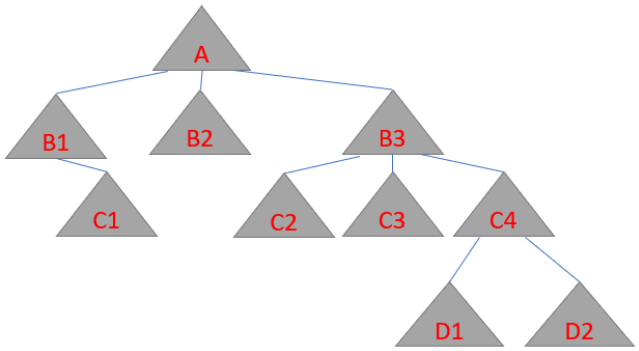


Fig.2. Tango tree T of fig.1(4).

II. Node in tango tree

For each node n in tango tree T, it has all attributes contained by the AVL tree: n.key, n.left, n.right, n.parent. and n.height. Here, height means the height of the node in the aux tree of T rather than in P.

There are some extra attributes:

n.depthP: depth of the corresponding node in P (not T !)

n.minDepth: the minimum of depthP of all nodes in its subtree including itself. This attribute is listed in the original paper but not required.

n.maxDepth: the maximum of depthP of all nodes in its subtree including itself

n.mark: whether the node is the root of an aux tree

When a leaf is attached to a marked node, the height of this leaf will not increase. When we update the height of parent aux tree or rotate/rebalance this parent aux tree, the attached children aux tree is equivalent to nil.

III. Operations of tango tree

Since data is reorganized in T, the operation in T is more complicated than P. We first understand the operation from the view of P, then discuss how to realize it in T.

A. from the perspective of P

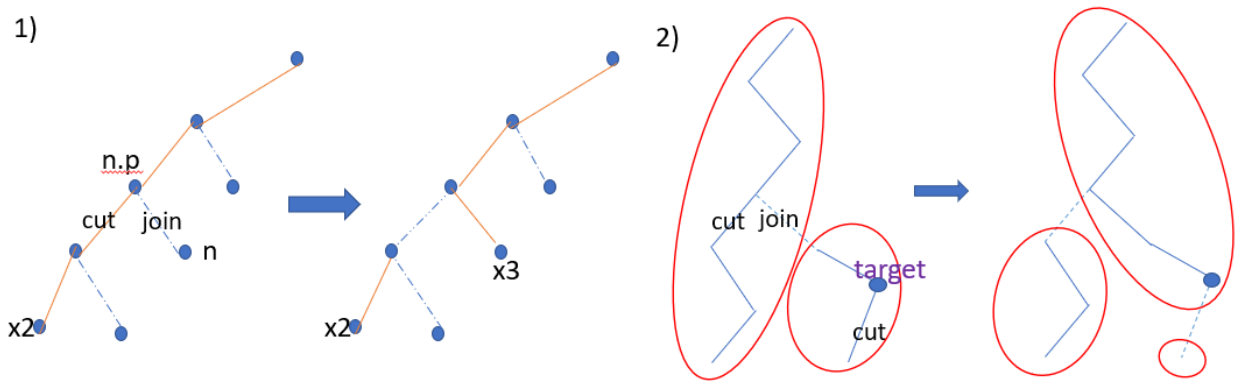


Fig.3. Search operation from view of P. 1)the change of preferred path from fig.1(3) to (4). 2)generalized case

In Fig.3(1), when x3 is accessed, the highlighted preferred path is cut once and join once. Fig.3(2) shows the general case. When new access is searched, the preferred path will change, which lead to the change of tango tree.

SearchTango(k): search key k

1. start from root, search the first aux tree, if find the node n with n.key==k, return n
2. else if reach nil, return nil
3. if reach a marked node n, it means the pointer reaches another aux tree
    1) split the first aux tree into two parts, top_aux and bottom_aux, the cut point is n.p: cut(T,n.p)
    2) unmark n and combine its aux tree with top_aux: join(T,n)
4. continue search, if meet another marked node, repeat step 3, until find key or reach nil, return n
5. if n has child, mark its child and split the aux tree into two parts: cut(T,n)

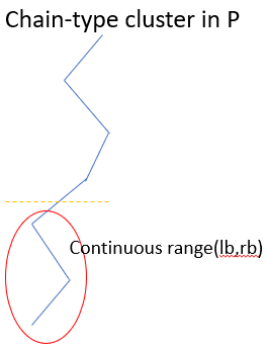
Cut(T,node n): cut an aux tree into two trees, one stores the nodes whose depth is greater than n.depth, another stores nodes whose depth<=n.depth

1) cutAtdepth(T, n.depth): cut the preferred path beneath n.
2) all nodes beneath n.p form a new aux tree and mark the new root

Join(T,node n):

1) unmark node n
2) connect the aux tree containing node n with the top aux tree

In P, these operations are straightforward. However, this chain-type cluster in P is compressed to AVL tree in T. We need more work to realize cut and join in T.



The core idea is to make use of the Continuous property for the chain-type tree.

Continuous property:

Due to the Inorder traversal, the nodes deeper than a certain depth d form a continuous range (lb,rb). Any nodes above this depth cannot fall into this range. In T, although these nodes are reconnected, we can split the nodes within (lr,rb) from the aux tree. This operation is equivalent to separate all nodes in P deeper than depth d.

B. from the perspective of T

The first step is to find the range(lb,rb), which is an open boundary. We need to find the closed boundary [l,r], then node lb is the predecessor of l and node rb is the successor of r.

minNodeDeeperThan(BiTreeNode n, int d): input depth d and tree rooted at n, find the **closed** lower boundary node l

1) Starting from the root n, if n.maxDepth<=d, it means n does not have child deeper than depth d, return null;
2) Else, if the pointer cannot go to left child (n does not have left child or left child is marked), then check the right child. If right child is not marked and right child's maxDepth>d, go to right child, return ***minNodeDeeperThan(n.right, d)***. Otherwise, return n;
3) The pointer can go to left child. Then return ***minNodeDeeperThan(n.left, d)***

The height of aux tree in T is O(loglogn), so this operation costs O(loglogn).

maxNodeDeeperThan(BiTreeNode n, int d): find the **closed** upper boundary

Similarly, the upper boundary r can be found by going through nodes on the right path.

Based on the closed boundary [l,r] obtained from previous methods, we can determine the open range (lb,rb), The next step is to cut the nodes within this range (lb,rb) as shown in Fig.4. The split and merge are bulk operations for AVL aux tree. We will discuss concrete operations later.

The split operation is to split one AVL tree into two AVL trees. All nodes smaller than k form an AVL tree, and nodes larger than k form another AVL tree. Split operation costs O(logn) for the tree with size n.

The merge operation is to concatenate two aux trees into one. Merge costs O(logn) for a tree with size n.

We will discuss split and merge of aux tree in the next section.

Here, because each aux tree has O(logn) nodes. So the merge and split cost O(loglogn) in the tango tree.

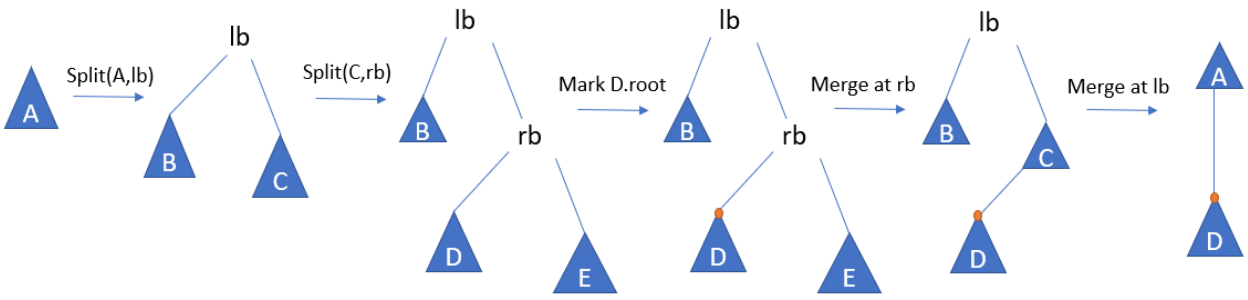


Fig.4. Cut aux tree in T. This figure is copied from ref[1].

cutWithinLR(lb, rb): the process is shown in Fig.4.

First, aux tree A is split by lb and form B and C, C is split by rb and form D and E. Then we know all nodes in D is within (lb,rb). We mark the root of D and form a new aux tree. Then merge the remaining trees into A. As a result, the A is split into A and D. D corresponds to the nodes deeper than depth d in P.

This operation has two split and two merge operations, cost O(4loglogn).

Join(tree A, tree B): //here A is higher than B in P, A and B are both chain-type in P. In T, A and B are AVL trees.

Due to the continuous property, all nodes in B must fall into the range of two neighbored nodes in A.

First, search the root of B in A, return n. if n.key<B, then B is within the range (n,n.successor).

If n.key>B, B is within the range (n.predecessor,n).

Second, once determine the range of B, reverse the steps of cut operation in Fig.4. Similarly, join operation costs O(4loglogn).

All these aforementioned operations depend on two bulk operations [2]: merge and split. We will show how to realize them in O(logn) time in AVL tree with size n.

IV. AVL aux Tree

For every internal node p in AVL tree, the heights of children of p differ by at most 1.

Insert or remove one element may break this height-balance property. Starting from the last touched node, we keep going up to its parent and check the height balance. When the height-balance is not conserved, do single-rotation or double-rotation to modify the structure. It takes O(logn) to search the unbalanced parent. So the insert and remove operation cost O(logn).

Here, the operation on aux tree is not to insert or remove one element. We need bulk operations to split and merge AVL tree. Ref[3] discusses these operations for RB tree, it can be easily extended to AVL tree.

Merge(t1,k,t2): nodes in t1 is smaller than k, nodes in t2 is larger than k. t1/t2 has height h1/h2.

1. If |h1-h2|<=1, set t1 as k's left child, t2 as k's right child, the new tree rooted at k is balanced
2. Otherwise, suppose h1-h2>1. Walk down to the right path of t1 until c with c.height==h2. set k as right child of c.parent, c as left child of k, t2 as right child of k. (Fig.5)
   An exception case: when go along the right path, may meet a node with node.height=h2+1 but node.right.height=h2-1, choose this node as c
   This step cost O(h1-h2)

3. Rebalance. Start from k and go up until root, it the parent node is unbalanced, do single or double rotation to rebalance it. This step cost O(h1-h2)
4. When h1-h2<-1, this condition can be done similarly.

In total, the merge operation cost O(|h1-h2|)=O(logn)

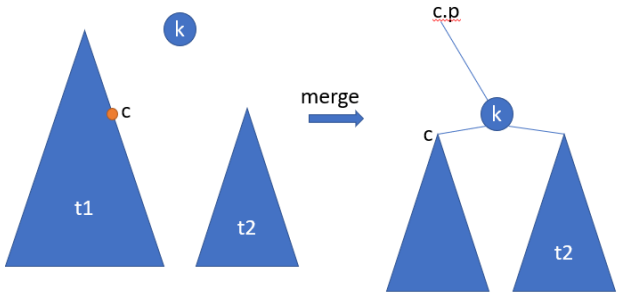


Fig.5. merge two AVL tree into AVL tree

Split(t,k): split tree t to two parts, so that all nodes in t1 is smaller than k, and all nodes in t2 is larger than k.

1. Search k in tree t. The search path split tree into many subtree connected by the highlighted search path. Search cost O(logn)
2. If root.key==k, then t1=root.left, t2=root.right
3. If root.key<k, go left and split the left subtree by k and obtain sub_left and sub_right. The new right tree is the combination of sub_right and root and root.right. The new left tree is sub_left. Likewise, if root.key>k, repeat step 3 symmetrically.
   The split process is a recursive progress, occurs at most O(logn) times.

In total, the Split operation cost O(logn).

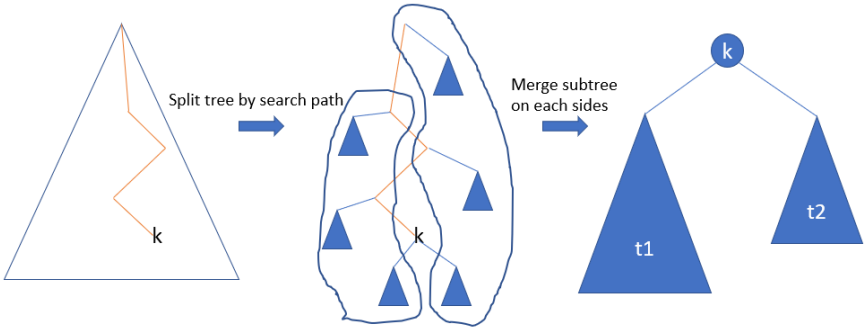


Fig.6. split AVL tree into two AVL trees

V. Cost analysis

There is no discussion about the amortized cost of the tango tree. Because the performance of the tango tree largely depends on the clustering rate of P tree.

1. In the original case such as in fig.1(1), all nodes are marked and each node is aux tree of themselves. In this case, search a node is the same as the BST, costs log(n).
2. Gradually access a series of keys, {x1,x2,…,xm}, this will lead to the evolution of tree from fig.1(1)-(4). As more and more nodes are searched, the nodes in P are increasingly clustered. When the number of searched keys m is around O(logn), the tree is largely clustered. There are not too many isolated nodes in T.
3. Now look for a node n.
    1) if n is in the top aux tree, it cost O(loglogn) to find it.
    2) If n is in the aux tree connected with the top aux tree, then search in the first tree cost O(loglogn). The pointer jumps from the first aux tree to second aux tree need two steps (Fig.3): cut and join, each cost O(4loglogn).
    Search through the second aux tree cost O(loglogn).
4. From step 3 we can conclude that the cost of search depends on which aux tree the node n is in. If we need do k jumps between aux trees, the total cost would be O((k+1)loglogn)

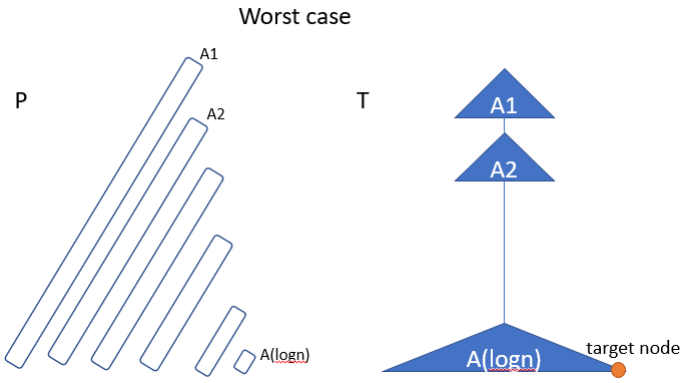


Fig.7. worst case for searching

Worst scenario: suppose there are O(logn) aux trees (fig.7) and the target nodes is in the bottom aux tree, then we need O(logn) times jumping to connect A1 with A(logn), which will cost O(logn*loglogn).

Normal scenario: suppose we already searched a specific node 100 steps before. Then the pointer needs at most 100 jump between aux tree to reach this node. In this situation, the tango tree cost O(100loglogn)=O(loglogn).

These two scenarios show the performance of tango tree. Even in the worst case, O(logn*loglogn) is also close to O(logn) generally. If the node is touched previously, then the cost time is O(loglogn), which is much better than RB tree or AVL tree.

When the system is small, there is not big difference between O(loglogn) and O(logn). In addition, the number of jumping k may be not small in tango tree, which worsens the real performance. So the tango tree may function as a supplementary method when the dataset is extremely large.

So far, there is no real application for the tango tree, it remains on the theoretical level to show the limit of the performance of the tree.

Other relevant sources [4–6,7,9].

Ps: There is a distributed system called Tango [8]. I think it is not relevant to the tango tree.

Appendix: Comparison between example and cost analysis

This work has three major operations: two bulk operations (split, merge) of AVL aux tree and search operation for tango tree.

A. The split operation for AVL aux tree

1. Generate a random AVL Tree with n nodes
2. Get the Inorder transversal of this AVL tree and pick the $\frac{n}{3}th$ node as the split node
3. Split the AVL tree by this split node and obtain two AVL tree, so that all nodes in the 1st tree are smaller than this split node and all nodes in the 2nd tree are larger than this split node

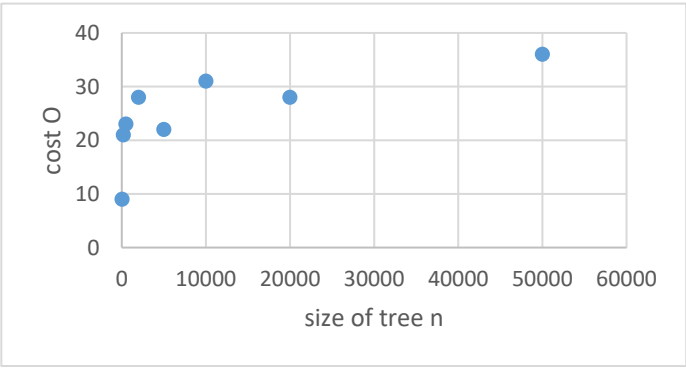
In code: AVLTree_split_test(int n)

Below is an example: split an AVL tree with size 50 and get 2 trees

```
original tree: The key list of tree (inorder):4971 5243 6621 13088 13274 13583 17073 17552 19066 23510 24070 27640
30358 31860 34943 35464 36042 37035 38325 40175 41796 42714 48805 50520 50539 50710 52243 54250 54271 58068 59928
62248 63185 63236 63373 63731 66164 68534 69818 70995 71453 80565 80585 82535 84978 85475 86764 91938 94625 98426
98674
size of original tree: 51
splited 1st tree: The key list of tree (inorder):4971 5243 6621 13088 13274 13583 17073 17552 19066 23510 24070
27640 30358 31860 34943 35464 36042
size of 1st tree: 17
splited 2nd tree: The key list of tree (inorder):38325 40175 41796 42714 48805 50520 50539 50710 52243 54250 54271
58068 59928 62248 63185 63236 63373 63731 66164 68534 69818 70995 71453 80565 80585 82535 84978 85475 86764 91938
94625 98426 98674
size of 2nd tree: 33
Cost Counter o:9
```

Result:

size of tree n	cost O
50	9
200	21
500	23
2000	28
5000	22
10000	31
20000	28
50000	36



In pervious analysis, cost=O(logn), it's consistent with real result.

B. Merge operation for AVL tree

1. Generate two AVL trees so that any nodes in 1st tree is smaller than any nodes in 2nd tree
2. Merge these two trees with a node between them, return a new combined AVL tree

In code: AVLTree_merge_test(int n1, int n2)

Example: combine AVL tree with size 50 and 100

The key list of 1st tree (inorder):14 23 54 73 73 73 79 90 100 115 140 158 169 239 243 244 259 274 287 313 317 361 368 382 413 416 424 436 464 478 482 494 511 529 555 602 674 703 713 731 763 770 794 818 838 848 895 909 944 991

size of 1st tree: 50

The key list of 2nd tree (inorder):1003 1005 1008 1054 1063 1064 1065 1065 1082 1091 1093 1097 1107 1116 1131 1136 1137 1138 1179 1182 1209 1217 1244 1264 1277 1285 1286 1292 1298 1302 1328 1335 1338 1355 1358 1376 1420 1426 1433 1434 1440 1446 1461 1462 1467 1469 1530 1533 1568 1569 1583 1586 1604 1605 1617 1625 1630 1640 1647 1656 1670 1673 1685 1698 1700 1702 1704 1704 1705 1736 1737 1741 1743 1745 1748 1755 1762 1770 1770 1800 1801 1803 1803 1830 1836 1837 1845 1849 1862 1895 1897 1908 1918 1928 1930 1937 1944 1953 1970 1979

size of 2nd tree: 100

connection node: 1500

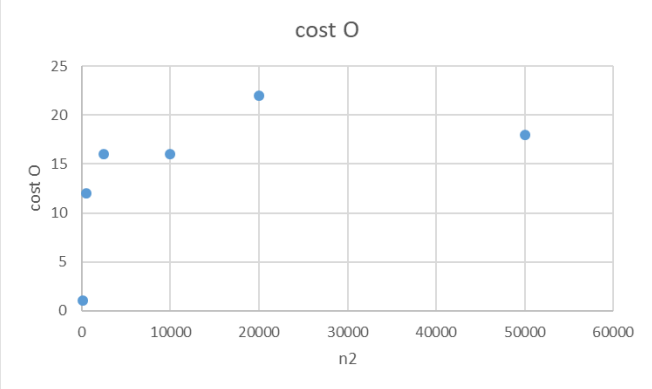
combined new tree: The key list of tree (inorder):14 23 54 73 73 73 79 90 100 115 140 158 169 239 243 244 259 274 287 313 317 361 368 382 413 416 424 436 464 478 482 494 511 529 555 602 674 703 713 731 763 770 794 818 838 848 895 909 944 991 1500 1003 1005 1008 1054 1063 1064 1065 1065 1082 1091 1093 1097 1107 1116 1131 1136 1137 1138 1179 1182 1209 1217 1244 1264 1277 1285 1286 1292 1298 1302 1328 1335 1338 1355 1358 1376 1420 1426 1433 1434 1440 1446 1461 1462 1467 1469 1530 1533 1568 1569 1583 1586 1604 1605 1617 1625 1630 1640 1647 1656 1670 1673 1685 1698 1700 1702 1704 1704 1705 1736 1737 1741 1743 1745 1748 1755 1762 1770 1770 1800 1801 1803 1803 1830 1836 1837 1845 1849 1862 1895 1897 1908 1918 1928 1930 1937 1944 1953 1970 1979

size of combined tree: 151

Cost Counter o:1

Result: Fix n1=50, increase the size of 2nd tree from 100 to 50000

size of tree n2	cost O
100	1
500	12
2500	16
10000	16
20000	22
50000	18



In previous analysis, cost =O(|h1-h2|)=O(logn), it is consistent with the result.

C. Search operation in tango tree

1. Generalize a random binary search tree with size n
2. Search a series node $\{node\ 1, node\ 2\}$. The tango tree is updated every time the target node is not in the top aux tree
3. Search $node\ 1$ again, it was touched two steps before, so the cost should be O(loglogn)

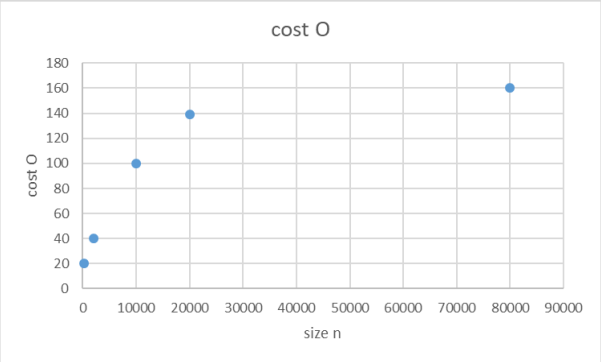
4. Search a new node3, which is not searched before, test the general performance

In code: TangoTree_search_test(int n)

Case 1: In step 3, when the target node was searched previously

Result:

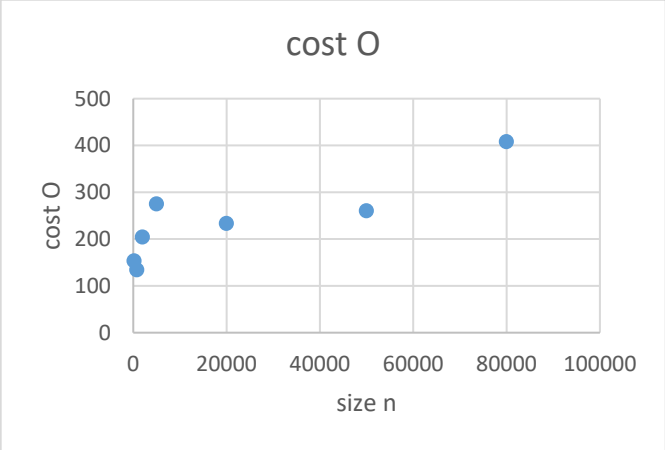
size of tree n	cost O
80000	160
20000	139
10000	100
2000	40
200	20



Case 2: in step 4, search a random node

Result:

size of tree n	cost O
200	153
800	134
2000	204
5000	275
20000	233
80000	408
50000	260



As expected, the behavior of tango tree in case 2 is worse than case 1.

When the system is not very large (such as n~10^4), there is no advantage for tango tree. Because the coefficient before O(loglogn) is very large in tango tree.

References

1. Demaine, E. D., Harmon, Dion., Iacono, John. & Paˇtraşcu, Mihai. Dynamic Optimality—Almost. *SIAM J. Comput.* **37,** 240–251 (2007).

2. AVL tree. *Wikipedia* (2019). at <https://en.wikipedia.org/w/index.php?title=AVL_tree&oldid=928494960>

3. Booth, H. D. An Overview over Red-Black and Finger Trees. (1992). at <http://citeseerx.ist.psu.edu/viewdoc/citations;jsessionid=E685DAEE37463FA0F811C8B554308BF6?doi=10.1.1.38.4454>
4. Bint, G. *kabourneeak/Tango*. (2014). at <https://github.com/kabourneeak/Tango>
5. Bose, P., Douïeb, K., Dujmović, V. & Fagerberg, R. An O(log log n)-Competitive Binary Search Tree with Optimal Worst-Case Access Times. in *Algorithm Theory - SWAT 2010* (ed. Kaplan, H.) 38–49 (Springer, 2010). doi:10.1007/978-3-642-13731-0_5
6. Hinz, T. F. Search Trees in Practice. 76
7. https://www.cs.princeton.edu/courses/archive/fall08/cos521/tango.pdf
8. http://www.cs.cornell.edu/~taozou/sosp13/tangososp.pdf
9. Junhui, D. Data structures and algorithms. (2005)

Code Fragments

The whole code: https://github.com/yuhao12345/AVL-based-Tango

class AVLTree

```
// merge two AVL tree and a boundary node b into a AVL tree,
//all nodes in Tree x < b.key, all nodes in Tree y > b.key, b is boundary node
// x,y here are root of tree
// for simplicity, b is not included by tree x or y !!!
// return root of new AVL tree x+b+y
public static BiTreeNode merge(BiTreeNode x,  BiTreeNode b, BiTreeNode y) {
        o++;
        if (x==null && y==null) {
                return b;
        }
        if (x==null && y !=null) {
                y.secede();
                // find left most node in y
                BiTreeNode lm=y;
                while(lm.hasLChild()) lm=lm.getLChild();
                lm.attachL(b);
                y=rebalance(lm,y);
                return y;
        }
        if (x!=null && y==null) {
                x.secede();
                BiTreeNode rm=x;
                while(rm.hasRChild()) rm=rm.getRChild();
                rm.attachR(b);
                x=rebalance(rm,x);
                return x;
        }
```

```
x.secede();
y.secede();
int h1=x.getHeight();
int h2=y.getHeight();
// |h1-h2|<=1, attach x to b's left side, attach y to b's right side, new tree is balanced
if ((h1-h2)<=1 && (h1-h2)>=-1) {
        b.attachL(x);
        b.attachR(y);
        return b;
}else if (h1-h2>1){
        // node c has same height as y
        BiTreeNode c=searchRightmostNodewithHeight(x, h2);
        BiTreeNode cp=c.getParent();
        c.secede();
        b.attachL(c);
        b.attachR(y);
        cp.attachR(b);
        // at least now nodes below cp is balanced, start from cp, go up until root x,
        //balance any node who dare to be unbalanced
        x=rebalance(cp,x);
        return x;
}else {  // symmetrical to the upper case
        BiTreeNode c=searchLeftmostNodewithHeight(y, h1);
        BiTreeNode cp=c.getParent();
        c.secede();
        b.attachL(x);
        b.attachR(c);
        cp.attachL(b);
        y=rebalance(cp,y);
        return y;
}
```

```
}
// split AVL tree to two AVL trees, all nodes in one tree are smaller than key b.key,
// nodes in another tree are larger than b.key
// for simplicity, boundary bode b can be found in tree r !!!
// return AVLTree left + node b + AVLTree right
public static ArrayList<BiTreeNode> split(BiTreeNode r, BiTreeNode b) {
        o++;
        BiTreeNode left=null,right=null,rl,rr;
        if (b==null) {
                left=r;
                right=null;
        }
        else {
                int k=b.getKey();
                ArrayList<BiTreeNode> a;
                if (r==null) {
                        left=null;right=null;
                }else{
                        if (r.hasLChild())
                                rl=r.getLChild();
                        else
                                rl=null;
                        if (r.hasRChild())
                                rr=r.getRChild();
                        else
                                rr=null;

                        if (rl!=null)
                                rl.secede();
                        if (rr!=null)
                                rr.secede();
```

```
                    if (k==r.getKey()){
                        left = rl;
                        right = rr;
                    }else if (k<r.getKey()) {
                        a = split(rl,b);
                        left = a.get(0);
                        right=merge(a.get(2),r,rr);
                    }else {
                        a = split(rr,b);
                        left=merge(rl,r,a.get(0));
                        right=split(rr,b).get(2);
                    }
                }
            }
            return new ArrayList<BiTreeNode>(Arrays.asList(left,b,right));
        }
```

class TangoAVLTree

```
    // in P tree, any nodes deeper (strictly larger) than depth d must fall in (lb2,rb2)
    // which is equivalent to cut nodes within the range (lb2,rb2)
    // first get closed boundary [lb,rb], than get (lb2,rb2)
    // r is the root of original top aux tree, return new root
    public BiTreeNode cutAtDepth(BiTreeNode r, int d) {
        BiTreeNode lb2,rb2;
        BiTreeNode lb=minNodeDeeperThan(r, d);  // for P
        BiTreeNode rb=maxNodeDeeperThan(r, d);
        if (lb==null) lb2=null;
        else lb2=lb.getPrev();
        if (rb==null) rb2=null;
        else rb2=rb.getSucc();
        return cutWithinLR(r,lb2,rb2);  // cut T
```

```
}


//in P tree, return minimum node deeper(>) than d.
// 1) if root.maxDepth<=d, it means no children in r is deeper than depth d, return null
// 2) else, if r does not have left child or left child is marked, go to r.right
// 3) r has left child, if its left child's maxDepth<=d, return r; otherwise go to r.left
public BiTreeNode minNodeDeeperThan(BiTreeNode r, int d) {
	if (r==null) return null;
	if (r.getMaxDepth()<=d) return null;
	if (!r.hasLChild()) {
		if (!r.hasRChild()) return r;
		if (r.getRChild().isMarked()) return r;
		if (r.getRChild().getMaxDepth()<=d) return r;
		r=r.getRChild();
	}else if(r.getLChild().isMarked() || r.getLChild().getMaxDepth()<=d){
		if (!r.hasRChild()) return r;
		if (r.getRChild().isMarked()) return r;
		if (r.getRChild().getMaxDepth()<=d) return r;
		r=r.getRChild();
	}else {
		r=r.getLChild();
	}
	return minNodeDeeperThan(r, d);
}
// in P tree, search max node deeper (>) than depth d
// find closed boundary n
public BiTreeNode maxNodeDeeperThan(BiTreeNode r, int d) {
	if (r==null) return null;
	if (r.getMaxDepth()<=d) return null;
	// now r.maxDepth>d
	if (!r.hasRChild()) {
```

```
                if (!r.hasLChild()) return r;
                if (r.getLChild().isMarked()) return r;
                if (r.getLChild().getMaxDepth()<=d) return r;
                r=r.getLChild();
        }else if (r.getRChild().isMarked() || r.getRChild().getMaxDepth()<=d){
                if (!r.hasLChild()) return r;
                if (r.getLChild().isMarked()) return r;
                if (r.getLChild().getMaxDepth()<=d) return r;
                r=r.getLChild();
        }else {
                r=r.getRChild();
        }
        return maxNodeDeeperThan(r, d);
}
// cut node within (lb,rb) from T tree, lb and rb are open boundary!!!
// r is root of top aux tree
public BiTreeNode cutWithinLR(BiTreeNode r, BiTreeNode lb,BiTreeNode rb) {
        ArrayList<BiTreeNode> a1,a2;
        BiTreeNode a,b,c,d,e;
        if (lb==null && rb==null) {
                return r;      // no nodes are cut from top aux tree
        }else {
                // r-> B + lb + C
                if (lb==null) {
                        b=null;
                        c=r;
                }else {
                        a1=AVLTree.split(r, lb);
                        b=a1.get(0);
                        c=a1.get(2);
                }
```

```
                // C -> D + rb +E
                if (rb==null) {
                        d=c;
                        e=null;
                }else {
                        a2=AVLTree.split(c, rb);
                        d=a2.get(0);
                        e=a2.get(2);
                }
                // mark root of D
                d.setMark(true);

                if (rb==null) {
                        lb.attachR(d);
                        a=AVLTree.mergeTreeNode(b,lb);
                        return a;
                }
                // rb + E -> C
                rb.attachL(d);
                // B + lb + C -> A
                if (lb==null) {
                        a=AVLTree.mergeNodeTree(rb, e);
                        return a;
                }
                c=AVLTree.mergeNodeTree(rb, e);
                a=AVLTree.merge(b,lb,c);
                return a;
        }
};
```

public BiTreeNode join(BiTreeNode a, BiTreeNode n) { // a is upper tree, b is lower tree

```
// 1st step, find range of n
//search n.root in a, get x. if x<n, than n is between (x,x.successor)
// Otherwise, n is between (x.predecessor,x)
BiTreeNode lb,rb; // left and right open boundary
BiTreeNode x=binsearchAux(a,n);
assert(x.getKey()!=n.getKey());
if (x.getKey()<n.getKey()) {
        lb=x;
        rb=x.getSucc();
}else {
        rb=x;
        lb=x.getPrev();
}
// 2nd step
ArrayList<BiTreeNode> a1,a2;
BiTreeNode b,c,e;
if (lb==null) {
        a1=AVLTree.split(a,rb);
        e=a1.get(2);
        n.setMark(false);
        return AVLTree.merge(n, rb, e);
}
if (rb==null) {
        a1=AVLTree.split(a, lb);
        b=a1.get(0);
        n.setMark(false);
        return AVLTree.merge(b,lb,n);
}
// now lb!=null && rb!=null
a1=AVLTree.split(a, lb);
b=a1.get(0);
```

```
            c=a1.get(2);
            a2=AVLTree.split(c, rb);
            e=a2.get(2);
            n.setMark(false); // unmark the root of n
            return AVLTree.merge(b,lb,AVLTree.merge(n,rb,e));
}
// search node n in top aux tree r
public BiTreeNode binsearchAux(BiTreeNode r, BiTreeNode n) {
            int k=n.getKey();
            if (r==null) return null;
            if (r.getKey()==k) return r;
            else if(k<r.getKey()) {
                        //search is within aux tree, when meet marked node,stop
                        if (!r.hasLChild()) return r;
                        if (r.getLChild().isMarked()) return r;
                        return binsearchAux(r.getLChild(),n);
            }else {
                        if (!r.hasRChild()) return r;
                        if (r.getRChild().isMarked()) return r;
                        return binsearchAux(r.getRChild(),n);
            }
}

// search in tango tree, the search itself will change the structure of tree when cross a marked node
// search the tree rooted at r
// change the tree structure and return the target node
public BiTreeNode searchTango(int key) {
            if (root==null) return null;
            BiTreeNode x=binsearchAux(root, key);
            if (x.getKey()==key) {
                        root=cut(root,x);
```

```
                        x.updateMaxDepth(); // important! update MaxDepth from the bottom of the newly
formed top aux tree
                        x.updateHeight();
                        return x;
                }
                if (x.getKey()>key) {
                        if (!x.hasLChild()) return null;
                        assert(x.getLChild().isMarked());
                        root=cut(root,x);
                        x.updateMaxDepth();
                        root=join(root,x.getLChild());
                        return searchTango(key);
                }else {// x.getKey()<key
                        if (!x.hasRChild()) return null;
                        assert(x.getRChild().isMarked());
                        root=cut(root,x);
                        x.updateMaxDepth();
                        root=join(root,x.getRChild());
                        return searchTango(key);
                }
        }
```