Binary Search Trees

Author Name:

Date: 2019-03-03

Chapter 1: Introduction

Problem description

In this project, two fixed insertion and deletion sequences are required for the three trees, same as a random insertion and deletion sequence. Compare and analize the three kind of tree structures through test analysis.

In the first and second case, the insertion sequence is N integers with increasing order. The deletion sequence is N integers with increasing order for the first case, and with decreasing order for the second case.

input:

- `N` is the size of the input integers
- The next N integers are inputted in the required order

background of the algorithms

In computer science,binary search tree is a tree structure in which each node has at most two branches. It is actually a particular type of container, which store items in memory. It allows fast lookup, insertion and deletion of items.

The time complexity of searching in a general binary search tree is related to the distance from the target node to the root of the tree. Therefore, when the depth of the node is generally large, the average complexity of searching will increase. And then, the balance tree came into being for more efficient in searching.

Unbalanced Binary Search Tree

Unbalanced binary tree is the most common binary tree structure that does not adjust the balance of a binary search tree. For an unbalanced binary search tree, inserting a node means adding this node to the leaf node of the tree. And there is no need to care about the balance of the tree after insertion and deletion.

AVL Tree

An AVL tree is a self-balancing binary search tree, which was the first such data structure to be invented. Insertions and deletions may require the tree to be rebalanced by one or more tree rotations, in which the newly-invented definition of balance factor is needed.

Splay Tree

A splay tree, which was invented by Daniel Sleator and Robert Tarjan is a self-adjusting binary search tree with the additional property that recently accessed elements are quick to access again. The most fundamental operation in this algorithm is called Splay, after which the tree will be rearranged to adjust a certain element to placed it at the root of the tree. There are two kinds of implementations as Top-down and bottom-up.

Chapter 2: Algorithm Specification

2.1 Main Data Structure

A basic data structure of binary search tree is widely used in this project. And for the AVL Tree and Splay Tree, a structure of self-balancing binary search tree is used. And they have a little difference with each other.

2.2 Unbalanced Binary Search Tree

2.2.1 Data Structure

```
typedef struct BSTNode *BSTTree;
struct BSTNode {
   BSTTree left;   /*left child*/
   BSTTree right;  /*right child*/
   int val;        /*value*/
};
```

2.2.2 Insert

The non-recursive way of insertion is complemented here to reduce the cost of memory. Firstly find the parent node of the node need to be inserted by comparing the value of the nodes. Then through value comparing to determine the new-node is whether the parent node's left child or right child. Finally, insert the new node.

```
void BSTInsert(BSTTree *T, int X)
{
    BSTTree ptr,P;
    point ptr to T;
    if (T is a NULL Tree) {
        Initialize new node;
        Make new node as the tree;
    }
    else
    {
        //find the node's parent node
        while (ptr point to non-null node)
        {
            P = ptr;
            if (X > ptr->val)
                ptr point to right child node
            else
                ptr point to left child node;
        }
        if (P->val > X){
            insert X node as left child node of P;
        }
        else if (P->val < X){
```

```
            insert X node as right child node of P;
        }
    }
}
```

2.2.3 Delete

When delete an integer X in the Binary Search Tree, it has following cases:

- When node with key X doesn't exist, we do nothing.
- When it is the only node of the tree, make the tree empty.
- When it is root and has no right or left subtree, make the tree be the right or left sub-tree.
- When it isn't root but has no right or left subtree, attach it's child node with it's parent node.
- When it has both left and right subtree, replace it with the minimum node in the right subtree.

```
void BSTDelete(BSTTree *Tree, int X)
{
    Find the node to be deleted and record its father;//it is similar with the steps in
insertion
    //First case:Doesn't exist node with key X
    if (node X not exist){
        printf("There's no node with this val\n");
    }
    else if (It is the only node of the tree){
        make the tree empty;
    }
    else if (It is root and has no left subtree){
        make the tree be the right sub-tree;
    }
    else if (It is root and has no right subtree){
        make the tree be the left sub-tree;
    }
    else{
        if (T has no left subtree)
        {
            Attach P and T's right subtree;
        }
        //T has no right subtree
        else if (T has no right subtree){
            Attach P and T's left subtree;
        }
        //T has both left and right child--Find the successor of T and exchange them
        else if(T has both left and right child){
            Find the minmum node as successor on the right side of T;
            Exchange T and its successor;
            Attach the deleted's right sub-tree with it's parent node;
        }
    }
}
```

2.3 AVL Tree

2.3.1 Data structure

```
typedef struct AvlNode *AvlTree;
struct AvlNode {
    AvlTree left;    /*left child*/
    AvlTree right;   /*right child*/
    int height;      /*height, and -1 for none node*/
    int val;         /*value*/
};
```

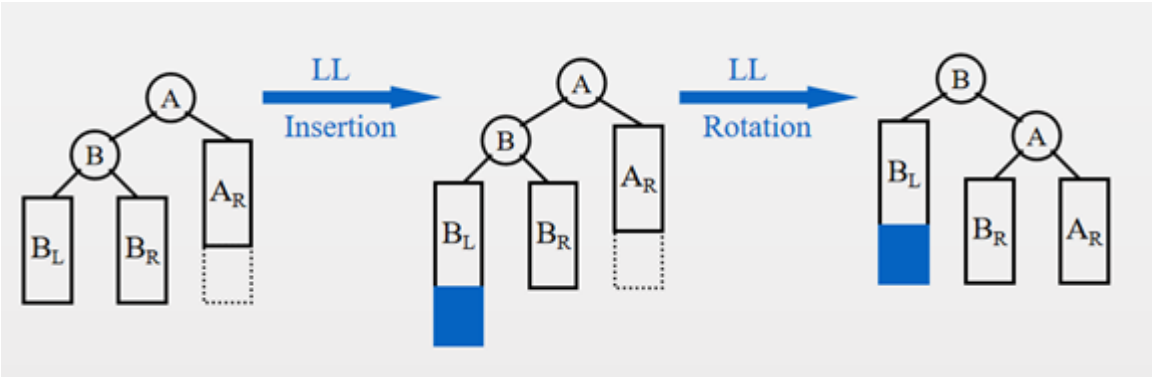
2.3.2 Insert

The insert operation in AVL tree is similar to the operation in Unbalanced BST. When insert an integer X to the AVL tree T, it has following cases:

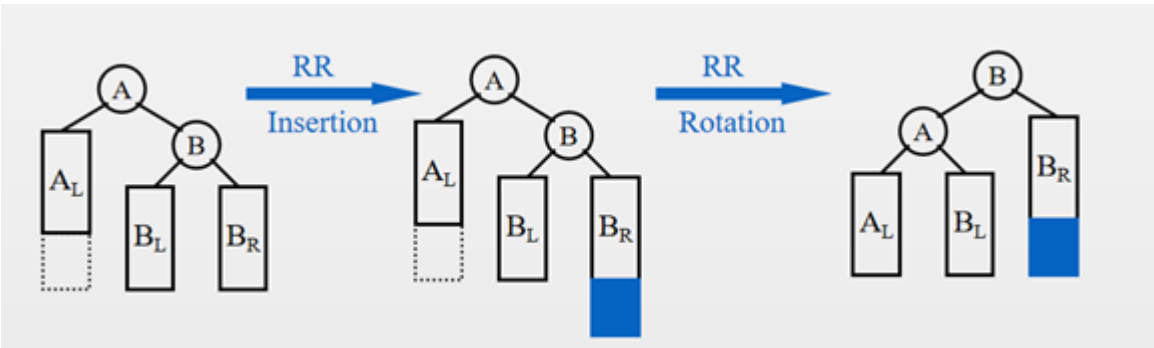
- When T is empty, we only need to create a new node and then initialize it.
- When X is smaller than the root's value, we recursively insert it into its left sub-tree.
- When X is larger than the root's value, we recursively insert it into its right sub-tree.

After Insertion, if the tree is unbalanced, there are four cases:

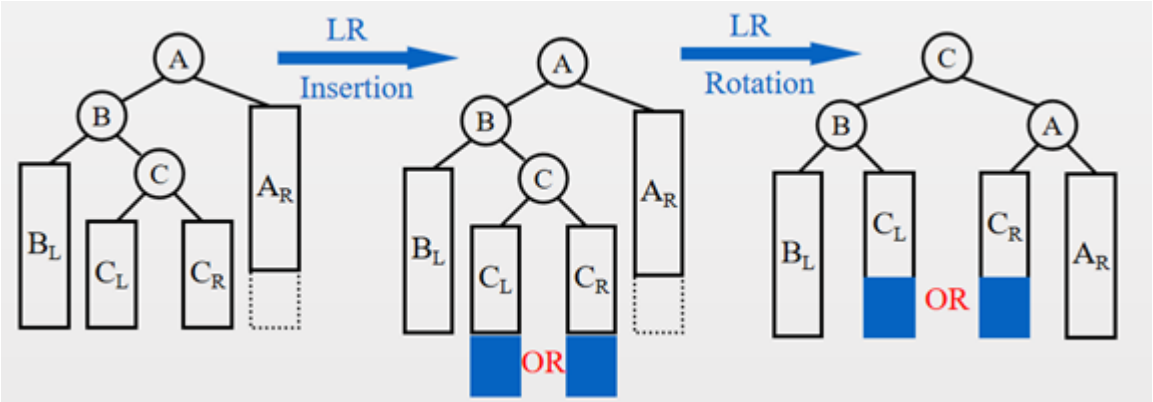
①New Node is in the Left - Left position of the root:



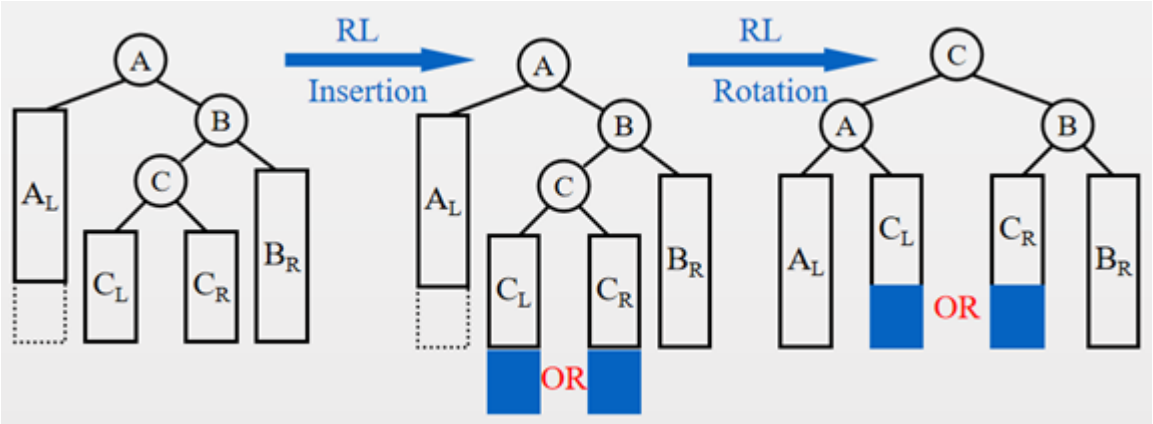
②New Node is in the Right - Right position of the root:



③New Node is in the Left - Right position of the root:



④New Node is in the Right - Left position of the root:



Finally, we need to update the height of the root.

The **pseudo-code** of insertion is as following:

```
AvlInsert(AvlTree T , int X) {
    if T == NULL                          /*Find the right position to insert*/
        T = malloc(sizeof(struct AvlNode);
        if T ≠ NULL                                                /*Initialize*/
            T->left = T->right = NULL;
            T->val = X;
    else if X < T->val
        T->left = AvlInsert(T->left , X);
        if HEIGET(T->left) - HEIGET(T->right) == 2    /*Tree is unbalanced*/
            if X < T->left->val
                T = AvlSingleRotateWithLeft(T);
            else
                T = AvlDoubleRotateWithLeft(T);
    else if X > T->val
        T->right = AvlInsert(T->right , X);
        if HEIGET(T->right) - HEIGET(T->left) == 2    /*Tree is unbalanced*/
            if X > T->right->val
                T = AvlSingleRotateWithRight(T);
            else
                T = AvlDoubleRotateWithRight(T);
    T->height = MAX(HEIGET(T->left) , HEIGET(T->right)) + 1; /*Update Height*/
    return T;
}
```

2.3.3 delete

The delete operation in AVL tree is difficult than the insert operation in AVL Tree. When delete an integer X in the AVL tree T, it has following cases:

- When T is empty, we do nothing.
- When X is smaller than the root's value, we recursively delete it in its left sub-tree.
- When X is larger than the root's value, we recursively delete it in its right sub-tree.
- When X is equal to the root's value, which means we find the node to delete, we need also to consider next three cases：

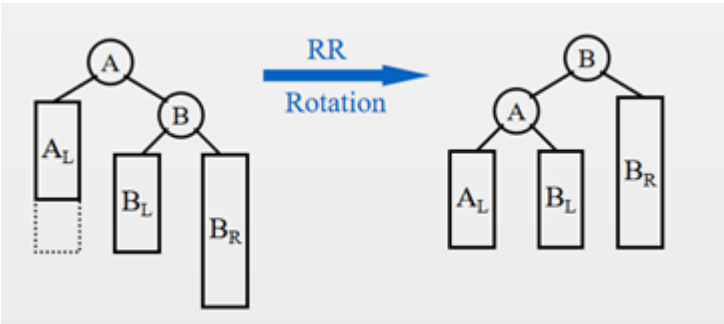
  ① If the node has no left child or right child, we only need to free it.

  ② If the node has only one child, we only need to update the root and then free the original node.

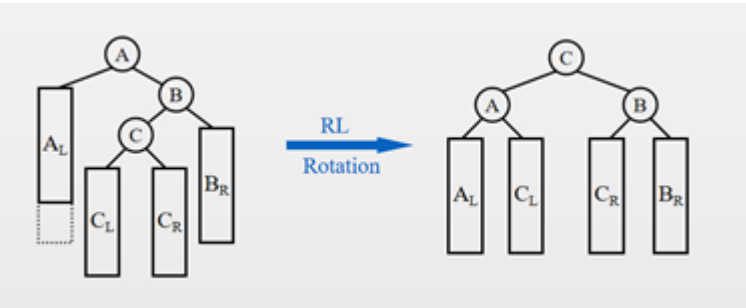
  ③ If the node has both left child and right child, similar to delete a node in the unbalanced tree , we need to find a node to replace the root. However, in order to make the tree balanced, it is necessary to compare its left sub-tree's height and right sub-tree's height first.When left sub-tree is higher than right sub-tree, we find the maximum node in the left tree and replace the root's value by it's value, then we delete in the left sub-tree recursively. And when left sub-tree is lower than right sub-tree, we find the minimum node in the right tree and replace the root's value by it's value, then we delete in the right sub-tree recursively. We know that delete a node at most decrease one level of its height, thus the tree is also balanced.

After delete the node, it is significant to check the tree is balanced in global. The following four cases will cause unbalanced:

① Deletion in the left sub-tree, right-right sub-tree higher than right-left sub-tree. As a matter of fact, its effect is equal to insert a new node in the right-right position of the root and make the tree unbalanced, thus we need to rotate the root with RR-Rotation.



② Deletion in the left sub-tree, right-left sub-tree higher than right-right sub-tree.



③ Deletion in the left sub-tree, left-right sub-tree higher than left-left sub-tree.

④ Deletion in the left sub-tree, left-left sub-tree higher than left-right sub-tree.

The **pseudo-code** of insertion is as following:

```
AvlDelete(AvlTree T , int X) {
    AvlTree TmpCell;
    if T ≠ NULL
        if X < T->val          /*Delete it in its left subtree recursively*/
            T->left = AvlDelete(T->left , X);
            if HEIGHT(T->right) - HEIGHT(T->left) == 2
                if HEIGHT(T->right->right) - HEIGHT(T->right->left) >= 0
                    T = AvlSingleRotateWithRight(T);
                else
                    T = AvlDoubleRotateWithRight(T);
        else if X > T->val   /*Delete it in its right subtree recursively*/
            T->right = AvlDelete(T->right , X);
            if HEIGHT(T->left) - HEIGHT(T->right) == 2
                if HEIGHT(T->left->right) - HEIGHT(T->left->left) > 0
                    T = AvlDoubleRotateWithLeft(T);
                else
                    T = AvlSingleRotateWithLeft(T);
        else
            if T->left ≠ NULL AND T->right ≠ NULL /*have both two children*/
                if HEIGHT(T->left) - HEIGHT(T->right) >= 0
                    AvlTree MaxNode = FindMax(T->left);
                    T->val = MaxNode->val;
                    T->left = AvlDelete(T->left , MaxNode->val);
                else
                    AvlTree MinNode = FindMin(T->right);
                    T->val = MinNode->val;
                    T->right = AvlDelete(T->right , MinNode->val);
            else                                    /*have at most one child*/
                TmpCell = T;
                T = (T->left ≠ NULL) ? T->left : T->right;
                free(TmpCell);
    return T;
}
```

2.4 Splay Tree

2.4.1 Data structure

```
typedef struct SplayNode *SplayTree;
struct SplayNode {
    SplayTree left;
    SplayTree right;
    int val;
};
```

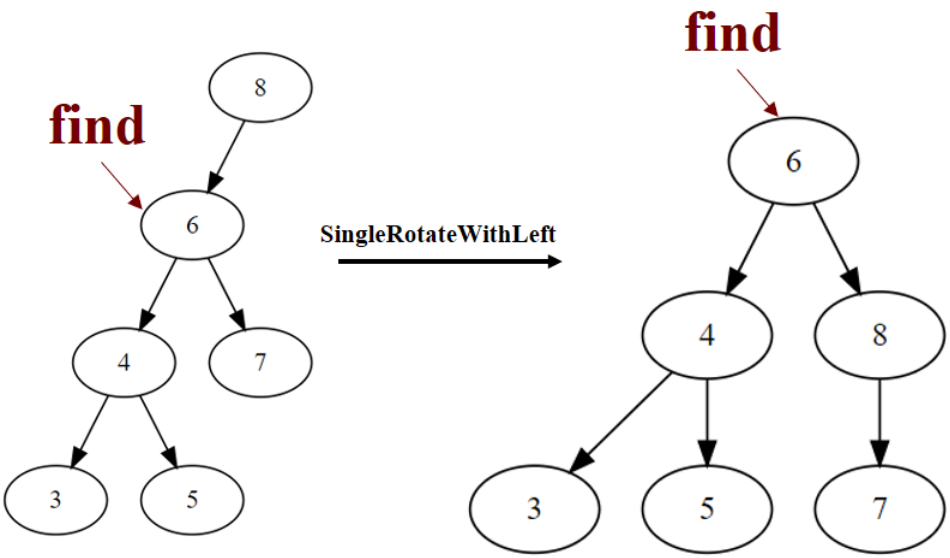
Algorithm 1: Bottom-up

This kind of algorithm firstly search for the X node from the top down. When searching for the X node, all nodes on the path from the X node to the root node are rotated, and finally the X node is replaced to the root, and depth of most nodes on the path is reduced.

2.4.2 Splay

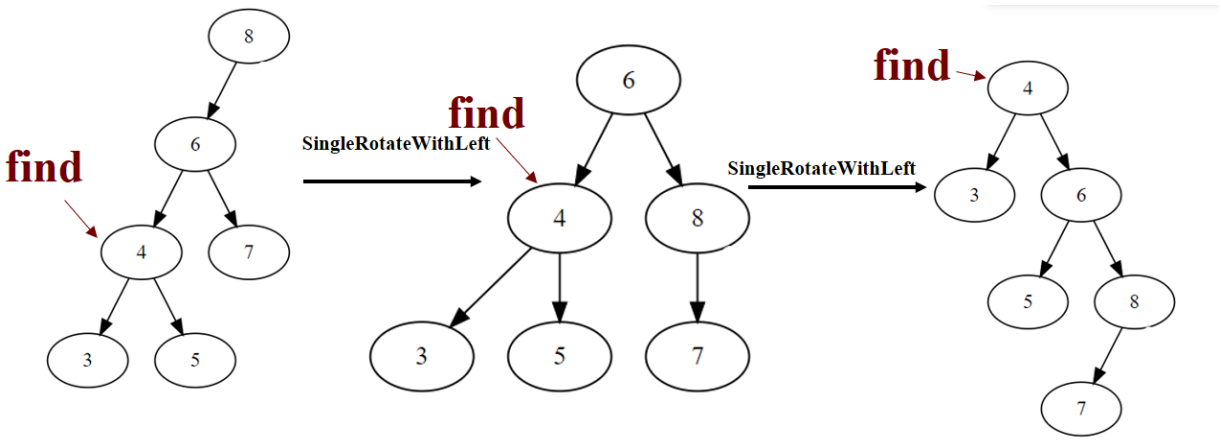
After the pointer pointed to the node with the value X, the tree is Splayed until the node is adjusted to the root node. Here we use a loop to adjust this node up. Following are some situations:

- This node doesn't have grand, as following case：



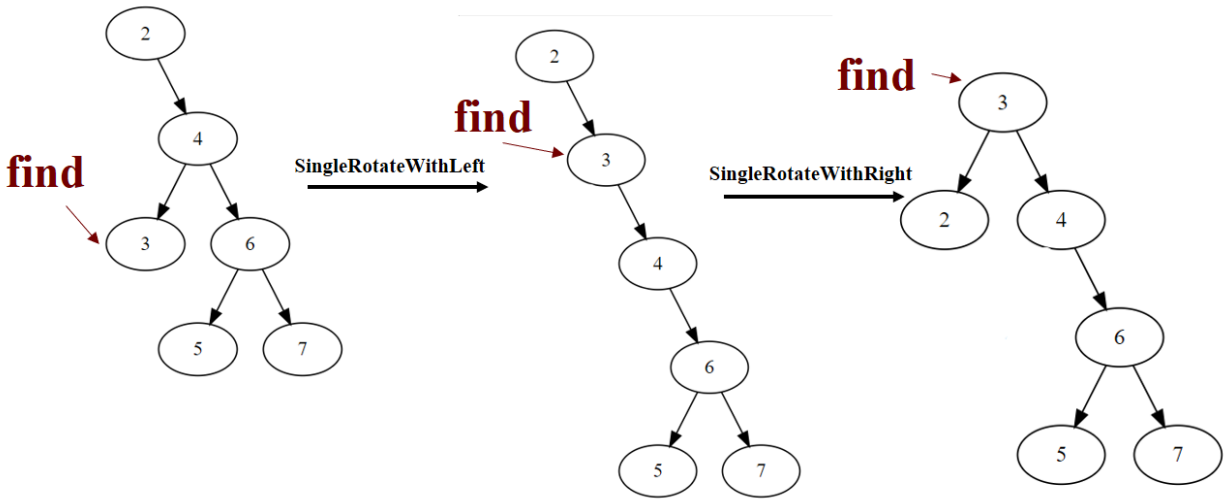
When the node doesn't have a grand-parent node, it only costs one left single rotate to adjust the node to the root node. This situation is exactly same with the X node is in the right subtree.

- The parent node of this node is the left node of the grand-parent node. We call this type of situation LL type, which is as following:



In this case it costs two times of left single rotate to adjust the node to the root node. This situation is exactly same with the X node is in the right subtree.

- The parent node of this node is the right node of the grand-parent node. We call this type of situation LR type, which is as following:



In this case it costs one double rotate to adjust the node to the root node. This situation is exactly same with the X node is in the left subtree.

```
SplayTree Splay(SplayTree X, SplayTree T)
{
    SplayTree P, G;
    P = X->parent;
    while (P not NULL) {
        if (X is the left child of P) {
            if (grand-parent is a NULL node)
                Single rotate P with left;
            else if (LL type) {
                Single rotate G with left;
                Single rotate P with left;
            }
            else if(RL type)
                Double rotate G with left;
        }
        else if(x is right child of P){
            if (grand-parent is a NULL node)
                Single rotate P with right;
            else if (RR type) {
                Single rotate G with right;
                Single rotate P with right;
            }
            else if(LR type)
                Double rotate G with right;
        }
        P = X->parent;
    }
    //X become the root of the tree
    return X;
}
```

2.4.3 Insert

Here, the insert operation is simple. You just need to find the right leaf node to put the node and Splay the tree to adjust the node to the root.

```
SplayTree SplayInsert(SplayTree T,int X)
{
    SplayTree NewNode;
    Initialize new node with its element equal to X;
    if (T has no nodes) T = NewNode;
    else {
        Attach the node with key X and its parent;
        splay the node to root;
    }
    return T;
}
```

2.4.4 Delete

It is similar with the delete operation in unbalanced Binary Search Tree. It has following cases:

- When node with key X doesn't exist, we do nothing.
- When it is the only node of the tree, make the tree empty.

When the node has been adjusted to the root:

- When it has no right or left subtree, make the tree be the right or left sub-tree.
- When it has both left and right subtree, find the maximum node in the left subtree, and Splay the left subtree to make the node as the root node of left subtree and then link the right sub-tree with it.

```
SplayTree SplayDelete(SplayTree T , int X) {
    SplayTree NewTree;
    if (T is not a NULL tree) {
        if (find(X)) {
            Splay the tree to make X as the root node;
            if (T have two subtrees)
                adjust the maximum node in left subtree to T->Left as newtree;
                link the right sub-tree with newtree;
            else if(T only has left subtree){
                delete the root node;
            }
            else if(T only has right subtree){
                delete the root node;
            }
            else if(it is the only node in tree){
                delete the node;
                return NULL Tree;
            }
        }
    }
    return T;
}
```

What's more

Later, we found that it is no need to adjust the tree when delete the node, because the main idea of adjust the node to the root is that the nodes have been accessed are more likely be accessed in the future. But in the case of deletion, this effect can be ignored. So the delete operation can be done in a more sufficient way by deleting it without adjust it to the root. Following is the pseudo-code, which is similar with the delete operation in Unbalanced Binary tree.

```
void SplayDelete2(BSTTree *Tree, int X)
{
    Find the node to be deleted and record its father;
    //First case:Doesn't exist node with key X
    if (node X not exist){
        printf("There's no node with this val\n");
    }
    else if (It is the only node of the tree){
        make the tree empty;
    }
    else if (It is root and has no left subtree){
        make the tree be the right sub-tree;
    }
    else if (It is root and has no right subtree){
        make the tree be the left sub-tree;
    }
    else{
        if (T has no left subtree)
        {
            Attach P and T's right subtree;
        }
        //T has no right subtree
        else if (T has no right subtree){
            Attach P and T's left subtree;
        }
        //T has both left and right child--Find the successor of T and exchange them
        else if(T has both left and right child){
            Find the minmum node as successor on the right side of T;
            Exchange T and its successor;
            Attach the deleted's right sub-tree with it's parent node;
        }
    }
}
```

But the first delete method still has its own advantage, for that when several nodes have been deleted, The balance of the tree may have a big change, which may need function Splay to adjust.

Algorithm 2: Top-down

This kind of algorithm is top-down, rotating while searching, and it can be completed by only one traversal on the access pass, which is higher in efficiency than in the bottom-up algorithm.

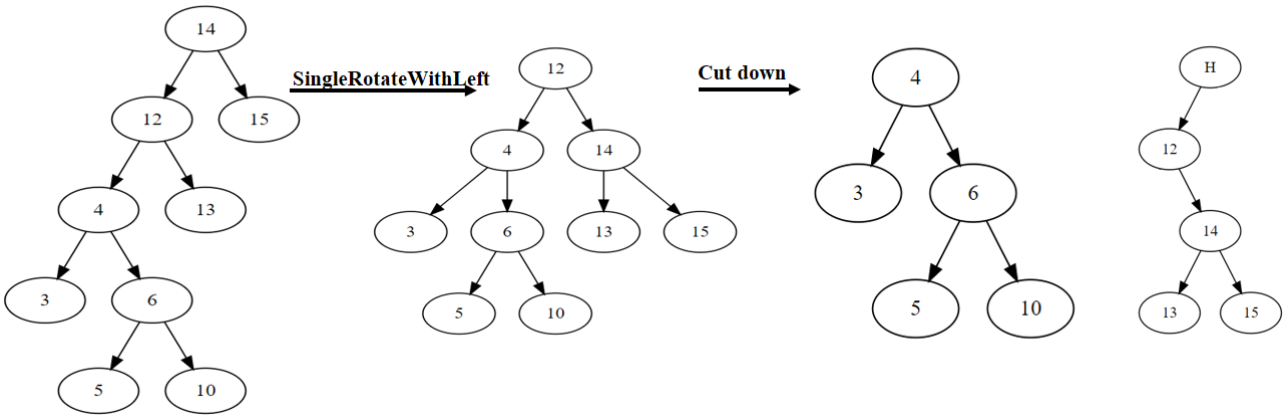
2.4.2 Splay

The Splay function uses X as the dividing line. If the X node exists in this tree, then the final adjusted tree has X as the root node, and the nodes in left sub-tree are all smaller than X, and the nodes in right sub-tree are all bigger than X. If the X node does not exist in this tree, then nodes in the left sub-tree of the final root node must be smaller than X, and nodes in the right sub-tree must be larger than X.

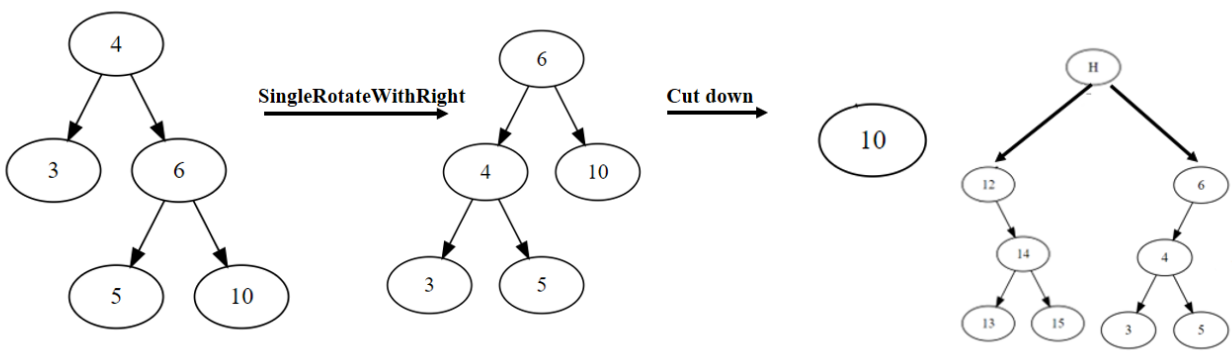
Firstly, build two empty trees and one node H. One is R and the other is L. The L tree holds all nodes smaller than X during the traversal process. H.left always points to the root node of the L tree and L always points to the largest node in the L during the traversal process. The same is true for R. The R tree holds all nodes bigger than X during the traversal process. H.right always points to the root node of the R tree and R always points to the largest node in the R during the traversal process.

Secondly, traverse from the root node T, find the X node until it is found or not found, during which there are 4 cases as following:

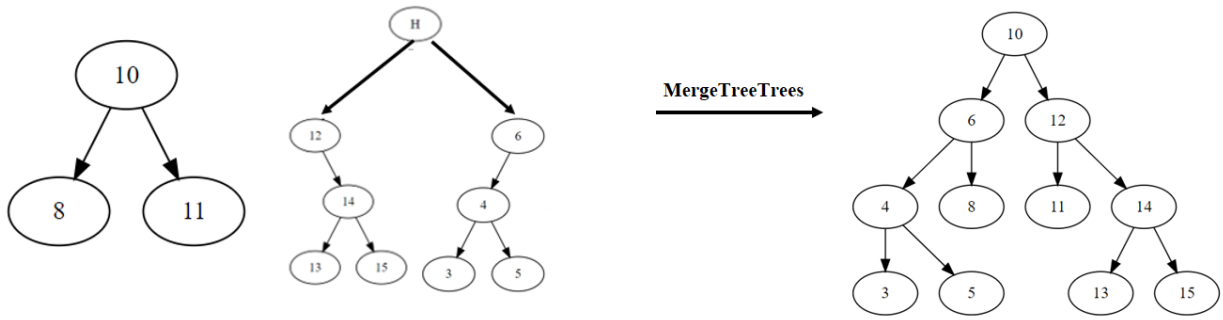
- If the X node is smaller than T and smaller than T->Left (LL type), a left single rotation around T is performed, and then T->Left becomes the new root node T'. X<T', so all nodes on the right subtree (including T') are larger than X. We hang T' on R tree and point R to the T' node, which is the smallest of all the nodes in R tree. Following is an example when find `X=5` :



- If the X node is less than T, but T->Left is NULL, the X node is not found, the loop is exited and the three sub-trees are merged.
- If the X node is greater than T and greater than T->Right (RR type), then a right single rotation around T is performed, such that T->Right becomes the new root node T', and X is greater than T', so all the nodes on the left subtree (including T') are smaller than X. We hang the T' subtree on L tree and point L to the new T' node, which is the biggest of all the nodes in L tree. Following is an example when find `X=10` :



- If the X node is bigger than T, but T->Right is NULL, the X node is not found, the loop is exited and the three sub-trees are merged. At this point L is the largest one less than the X node, so hang the M->Left of the neutron tree on L->Right, and hang M->Right on R->Left. Finally, update M to the root node of L and R tree.




```
SplayTree Splay(SplayTree T , int X) {
    SplayNode H as the header node;
    SplayTree L , R;   //L stores the node always less than X, R stores the node always
larger than X
    L = R = &H; H.val = X;  //initialize
    while (the element of the present node is not equal to X) {
        if (X < the present node) {
            if (X < the left child node)
                Single rotate T with left;
            if (T has no left child) break the loop;
            Separate the right sub-tree;
            link with R;
            keep looking in the left sub-tree;
        } else {
            if (X > the right node)
                Single rotate T with left;
            if (T has no right child) break the loop;
            Separate the left sub-tree;
            link with L;
            keep looking in the right sub-tree;
        }
    }
    link the left node to the L;
    link the right node to the R;
    reverse the left and right tree
    return T;
}
```

2.4.3 Insert

Firstly, create a new node to represent the newly inserted node. Then the value of the new node is used as the dividing value to perform Splay processing on the original tree. Get a new tree with the new node as the root node.

```
void InsertSplayTree SplayInsert(SplayTree T , int X) {
    SplayTree NewNode;
```

```
    Initialize new node with its element equal to X;
    if (T has no nodes) T = NewNode;
    else {
        Splay the tree T with X;
        if (X < root node of T) {
            link NewNode->left with left sub-tree,others as right sub-tree;
        } else if (X > root node of T) {
            link NewNode->right with right sub-tree,others as left sub-tree;
        }
    }
    return T;
}
```

2.4.4 Delete

The value of the node to be deleted is used as the dividing value to perform Splay processing on the original tree. Get a new tree with the node to be deleted as the root node. Then, there are two cases:

- The root node's value doesn't equal to X. Then terminate the procedure.
- T doesn't have left subtree. Then delete the root node.
- T have left subtree. Splay the tree T->left with X to make the right subtree empty and link the right sub-tree with new tree.

```
SplayTree SplayDelete(SplayTree T , int X) {
    SplayTree NewTree;
    if (T is not a NULL tree) {
        Splay the tree T with X;
        if (X in the tree) {   //x is already in the tree that can be deleted
            if (T have no left subtree)
                delete the root node;
            else {
                Splay the tree T->left with X to make the right subtree empty;
                link the right sub-tree with new tree;
            }
            free the old root and update the new root;
        }
    }
    return T;
}
```

Chapter 3: Testing Results

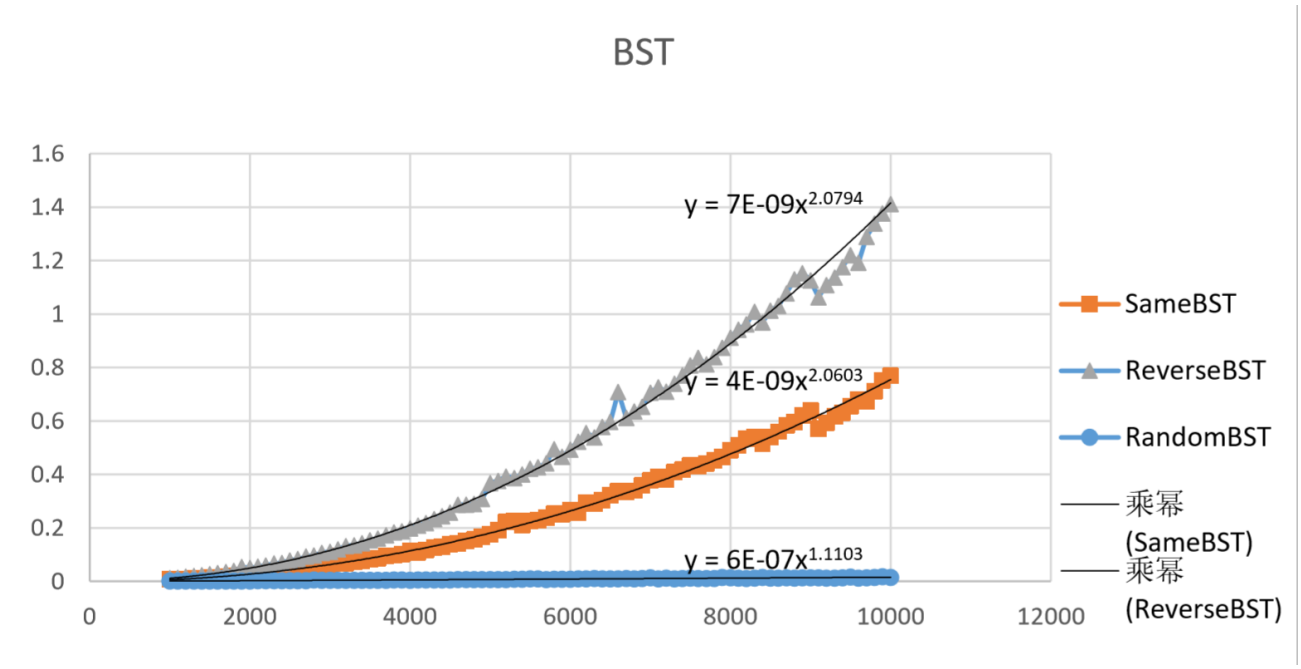
3.1 Run time table

Table 1 Run Time Table (Unit:0.001s)

Tree\N	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000
Way1： Insert in an increasing order and delete in the same order										
BST	8.14	27.08	59.399	113.84	176.5	266.62	377.18	491.44	638.64	768.9
AVL	5.883	12.454	19.426	26.44	34.105	43.946	52.239	60.112	67.027	72.9
TSplay	1.171	2.278	3.541	4.512	5.74	6.873	8.134	9.728	11.685	11.39
BSplay	3.86	3.18	5.424	7.72	10.65	10.125	10.929	13.333	14.273	16.9
Way2： Insert in an increasing order and delete in the reverse order										
BST	11.91	50.56	111.26	198.28	368.35	493.49	706.12	911.6	1126.2	1411.3
AVL	5.776	12.544	20.354	26.224	37.675	44.633	51.88	60.984	66.523	72.79
TSplay	1.085	2.38	3.21	3.968	5.53	6.813	8.007	9.008	10.234	10.83
BSplay	1.27	2.96	5.03	6.6	6.15	7.063	8.643	9.417	10.818	11.8
Way3： Insert in random order and delete in random order										
BST	1.2	2.42	4.775	4.96	7.15	8.434	14.296	13.52	13.604	15.7
AVL	6.485	14.308	23.051	29.044	39.99	49.602	61.81	76.832	79.198	87.99
TSplay	2.329	5.078	10.324	10.136	14.515	17.524	22.085	25.624	28.784	31.39
BSplay	4.81	11.58	15.939	30.32	28.25	35.562	46.786	55	63.636	72.2

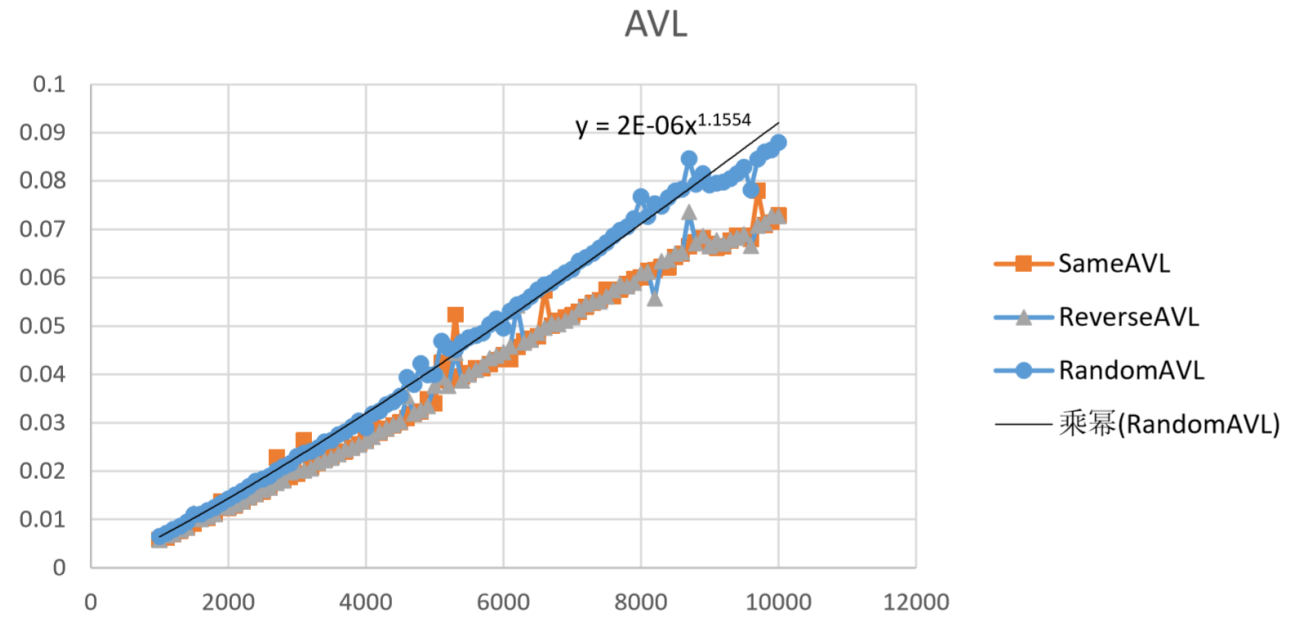
3.2 Different operations on the same tree （Unit:1s)

3.2.1 Unbalance BST



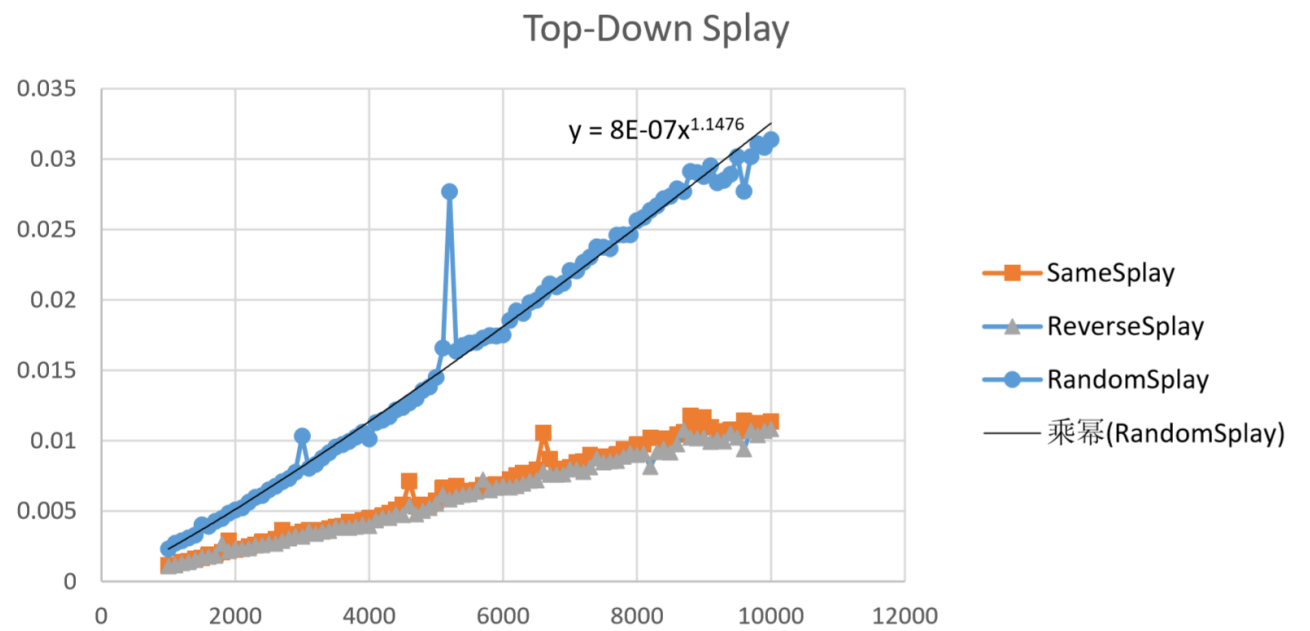
We can guess from the figure that the time complexity of the same order of insertion and deletion is about $O(N^2)$, the time complexity of the reverse order is about $O(N^2)$ and for random sequence is about $O(N^{1.1})$ or $O(N \log N)$.

3.2.2 AVL Tree



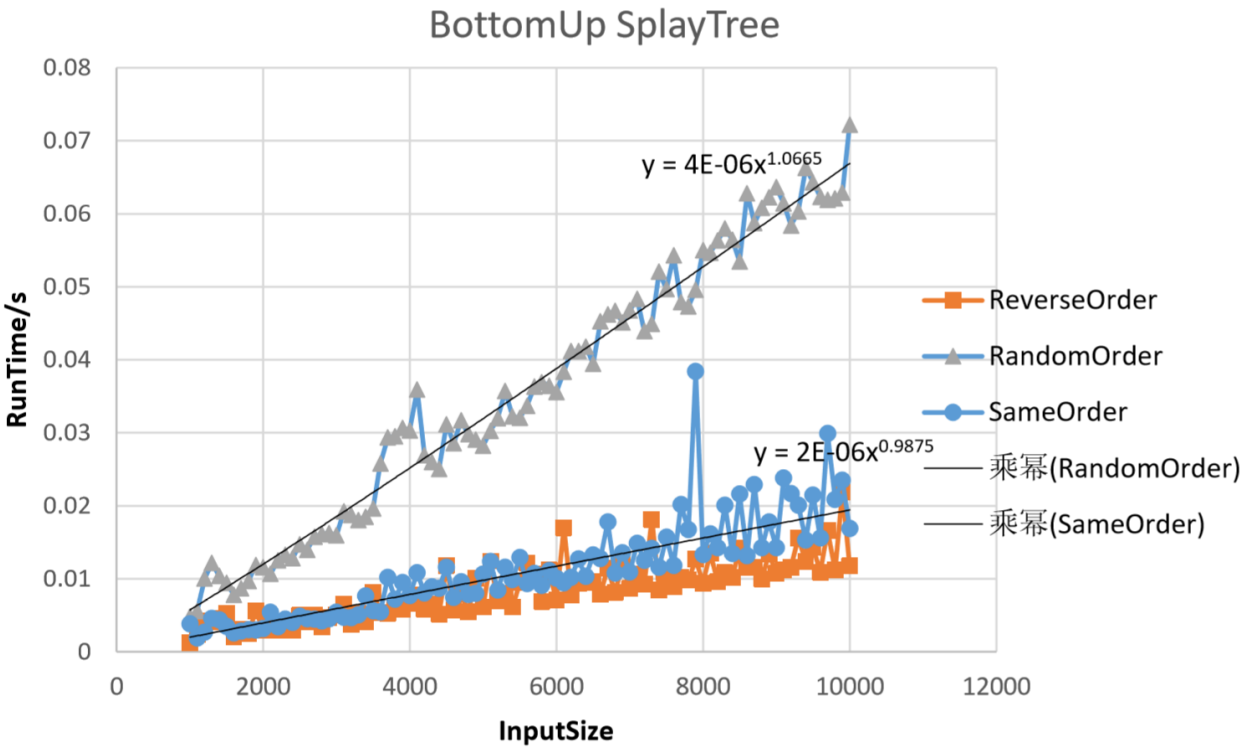
We can guess from the figure that the time complexity of the same order of insertion and deletion is about $O(N^{1.1})$ or $O(N \log N)$, the time complexity of the reverse order is about $O(N^{1.1})$ or $O(N \log N)$ and for random sequence is about $O(N^{1.1})$ or $O(N \log N)$.

3.2.3 Top-down Splay Tree



We can guess from the figure that the time complexity of the same order of insertion and deletion is about $O(N)$, the time complexity of the reverse order is about $O(N)$ and for random sequence is about $O(N^{1.1})$ or $O(N \log N)$.

3.2.4 Bottom-up Splay Tree

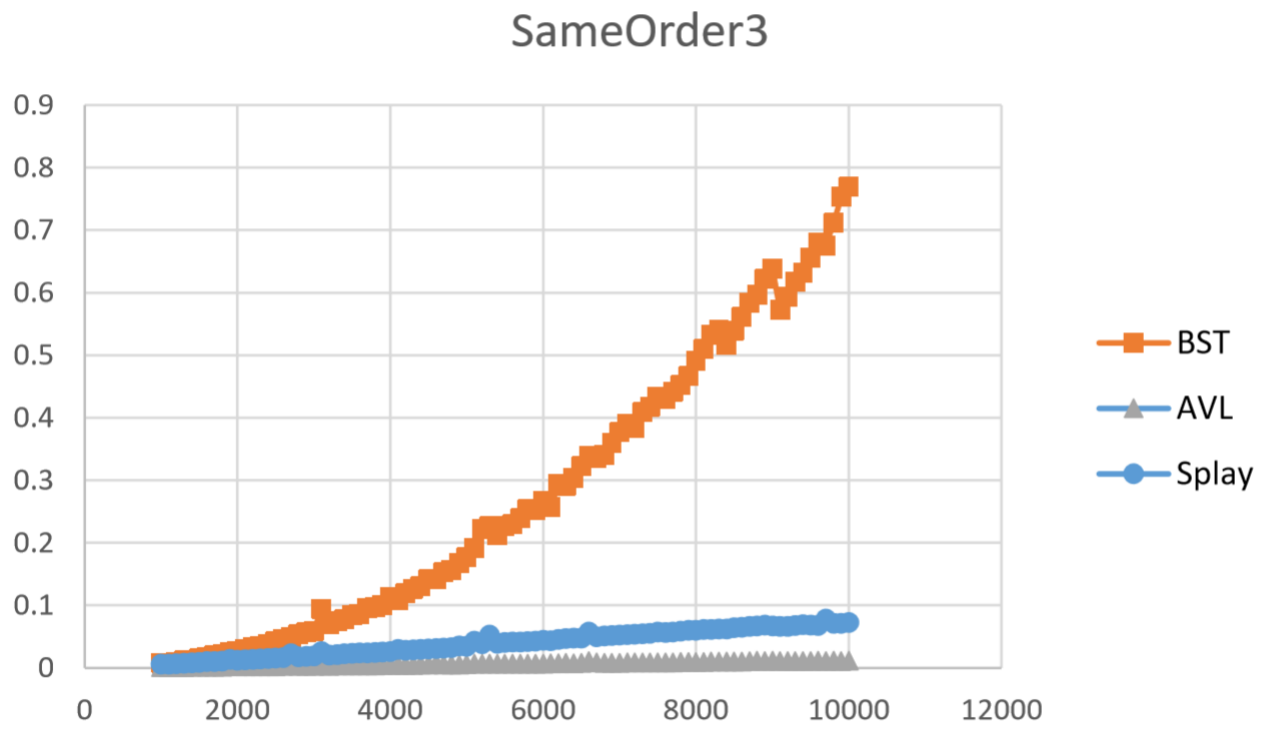


We can guess from the figure that the time complexity of the same order of insertion and deletion is about $O(N)$, the time complexity of the reverse order is about $O(N)$ and for random sequence is about $O(N^{1.1})$ or $O(N \log N)$.

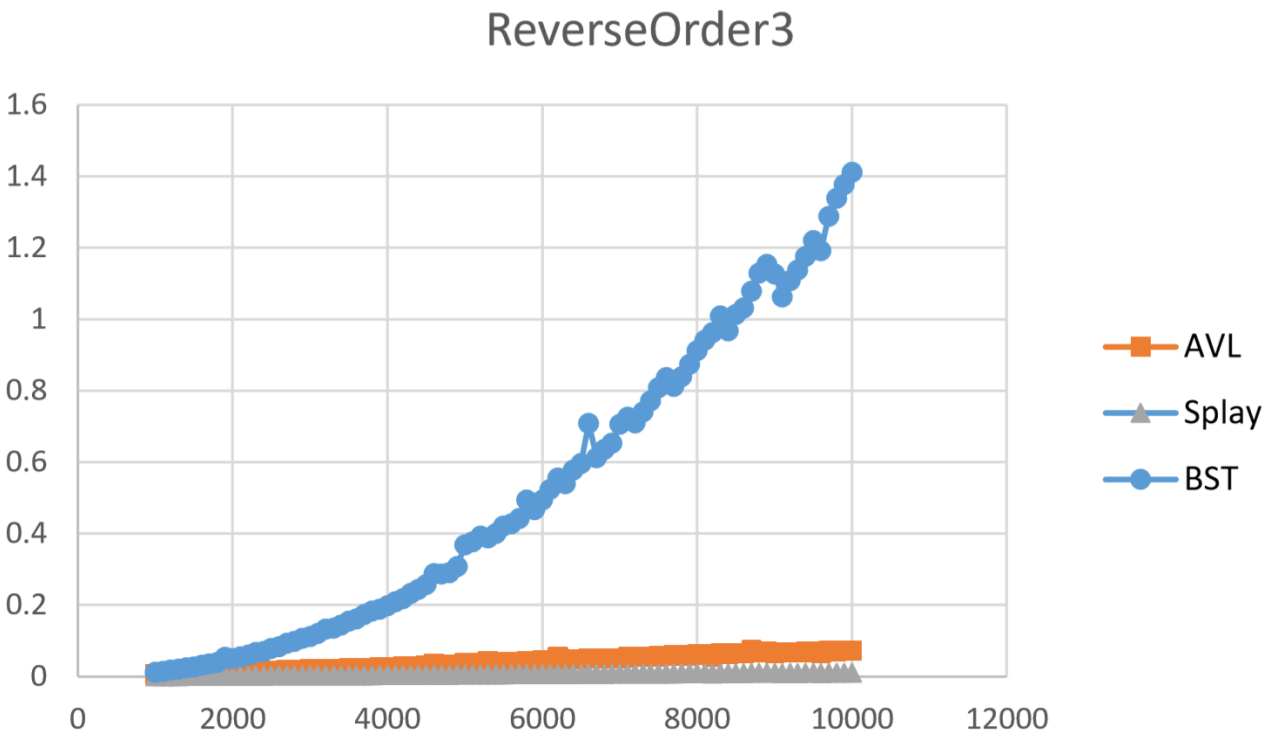
3.3 The same operation on different trees

As Up-down splay tree is more frequently used than Bottom-Up splay tree, we use Up-down splay tree's data to represent Splay Tree.

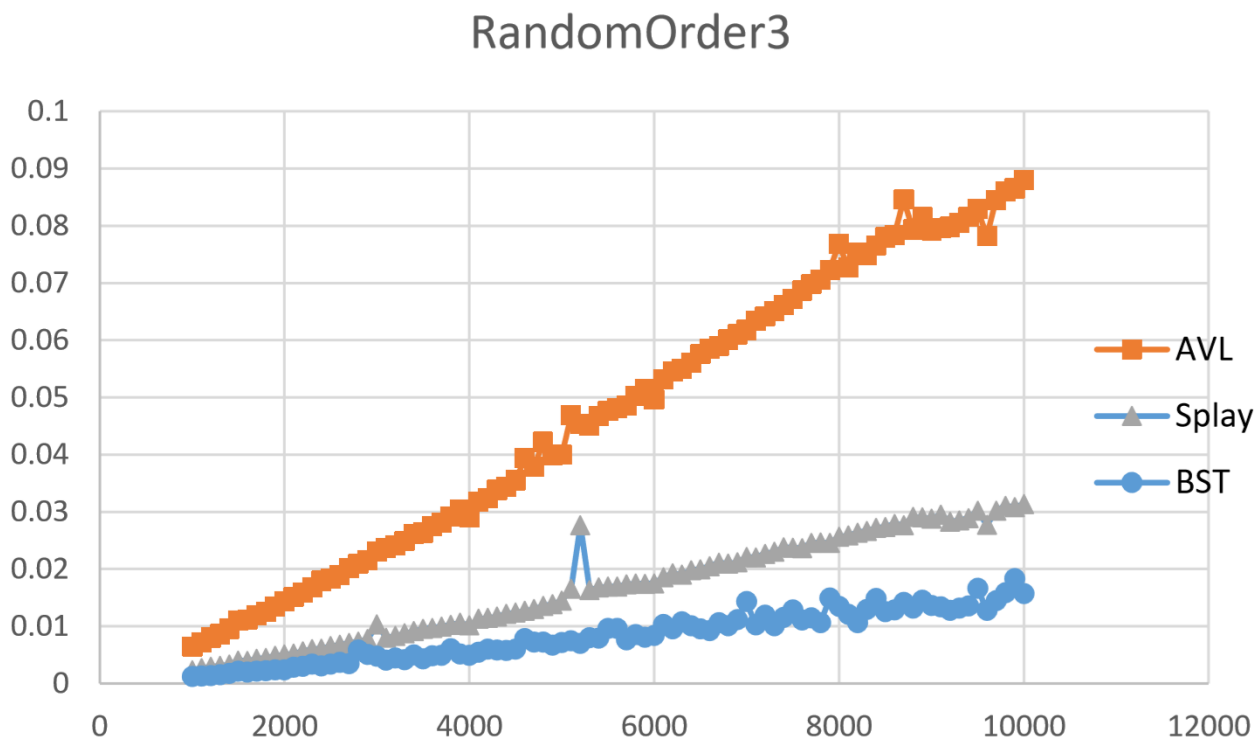
3.3.1 Insert in an increasing order and delete in the same order



3.3.2 Insert in an increasing order and delete in the reverse order



3.3.3 Insert in random order and delete in random order



3.4 conclusion

In **3.3.1** & **3.3.2**, the BST is just a chain and always a chain. Splay tree shows its strength on this condition since it can almost halves the depth of nodes on the path so the running time is much lesser than BST. In **3.3.3**, BST performs better than other two type of trees. We account this phenomenon for their strengths and weaknesses. AVL and Splay are all binary tree and they are designed for better searching, so they sacrifice some time when inserting and deleting. For example, AVL 's rotation costs a lot since it need to maintain its property. In **3.3.3**, we have no search operation without changing the structure of the tree so we can't see the strength of AVL and splay over BST. Tables and pictures plotted above can tell us the probable time complexity of each operation on each tree, but we can't tell whether the linear is $O(N)$ or $O(NlogN)$ with these lines only, so we will analyze it precisely in **Chapter 4**, calculating the true result.

Following are the advantages and disadvantages of the tree trees.

3.4.1 AVL tree

advantage

- The time complexity of find, insert and delete are all $O(\log N)$ at the worst case.
- Operations are simple.

disadvantage

- With the help of the balance factor, the element structure needs to be modified for this purpose.
- The cost of Rotating after insertion and deletion is expensive. The number of times of rotation up to $O(\log N)$ times after the operation of deletion.
- One of a series of lookup frequent insert or delete operations do not pay off.

3.4.2 Splay tree

advantage

- Comparable performance: Average-case performance is as efficient as other trees such as AVL tree.
- Small memory footprint: Splay trees do not need to store any redundant data.

disadvantage

- The height of a splay tree can be linear. For example, this will be the case after accessing all n elements in non-decreasing order.
- The representation of splay trees can change even when they are accessed in a 'read-only' manner. This complicates the use of such splay trees in a multi-threaded environment.
- Sometimes, a certain lookup operation can be time-consuming, which can be a disadvantage in a real-time application.

3.4.3 Unbalanced BST

advantage

- The operations are simple.

disadvantage

- The time complexity of find, insert and delete can be $O(\log N)$ in the worst case. And in the real life, it is usually that the data has been accessed is more likely to be accessed in the future, which can lead to a problem in unbalanced binary search tree.

Chapter 4: Analysis and Comments

4.1 Unbalanced Binary Search Tree

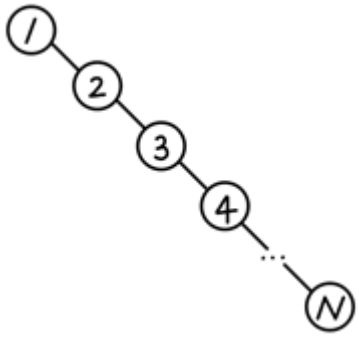
4.1.1 Insert N integers in increasing order and delete them in the same order

Assume that N integers are from 1 to N, after the procedure of insert N numbers, the binary search tree may like the following figure. When insert number 1, we need 1 time to find the right position, and when insert number 2, we need 2 times to find the right position, thus we can conclude that insert number N we need N times to find the right position. So we can calculate the total time of insert N integers in increasing order:

$$T = \sum_{i=1}^{N} i = O(N^2)$$

Similarly, when delete 1, we only need to transform the original root to its right sub-tree and then free it which cost only 1 time, so the total time of delete N integers in the same order is:

$$T = \sum_{i=1}^{N} 1 = O(N)$$



4.1.2 Insert N integers in increasing order and delete them in the reverse order

The operation insert is same as above, we only talk about delete them in the reverse order. First we delete number N, which costs N times, then we delete number $N-1$ and it costs $N-1$ times accordingly..... Finally, we expense 1 time to delete the last node 1. So the total time of delete N integers in ther everse order is :

$$T=\sum_{i=N}^{1} i=O(N^2)$$

4.1.3 Insert N integers in random order and delete them in random order

let $D(N)$ is the internal path length of the tree, assume its left sub-tree has i nodes and its right sub-tree has $N-i-1$ nodes, the next conclusion can be proved:

$$D(N)=D(i)+D(N-i-1)+N-1=\frac{2}{N}[\sum_{j=0}^{N-1} D(j)]+N-1=O(\log N)$$

Thus, insert N integers in random costs average $O(N\log N)$ time.

When it comes to delete N integers in random, the costs is also $O(N\log N)$.

4.2 AVL Tree

Because the time complexity of updating the height of the trees and rotation are both in $O(1). So the insertion and deletion operation time of the AVL tree mainly depends on the number of times of searching.

4.1.1 Insert N integers in increasing order and delete them in the same order

When insert number $i(i\geq 2)$, the height of the tree is $\lfloor \log i-1 \rfloor$, so we need $2+\log(i-1)$ time for every insert operation, thus the total time is :

$$1+\sum_{i=2}^{N}[\log(i-1)+2]=2N-1+\log((N-1)!)=O(N\log N)$$

When delete number i in the same order, because i is the minimum number in the tree, thus the height of the tree is $\log(N-i-1)$, which takes $\log(N-i-1)$ time to find, thus the total time is:

$$1+\sum_{i=2}^{N}[\log(i-1)+2]=N+\log N!=O(N\log N)$$

4.1.2 Insert N integers in increasing order and delete them in the reverse order

Insert is same as above. When delete number i in the reverse order, because i is the maximum number in the tree, thus the height of the tree is $\log(N-i-1)$, which takes $\log(N-i-1)+1$ time to find, thus the total time is also:

$$1+\sum_{i=2}^{N}[\log(i-1)+2]=N+\log N!=O(N\log N)$$

4.1.3 Insert N integers in random order and delete them in random order

Similar to the above analysis, the sequence here is a random sequence. Again, the height of the AVL Tree here is also strictly $\lfloor \log i-1 \rfloor$. So the time complexity of each operation is also $O(\log N)$ .So the total time is about:

$$1+\sum_{i=2}^{N}[\log(i-1)+2]=N+\log N!=O(N\log N)$$

It is similar for the delete operation, also because the tree height is $O(\log N)$ .So the total time complexity is:

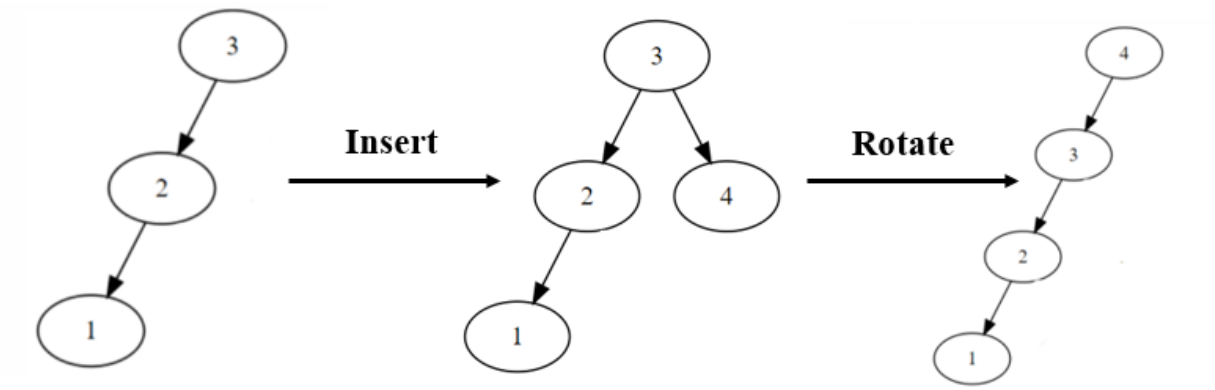
$$1+\sum_{i=2}^{N}[\log(i-1)+2]=N+\log N!=O(N\log N)$$

4.3 Splay Tree

Algorithm 1: Bottom-up

4.3.1 Insert N integers in increasing order and delete them in the same order

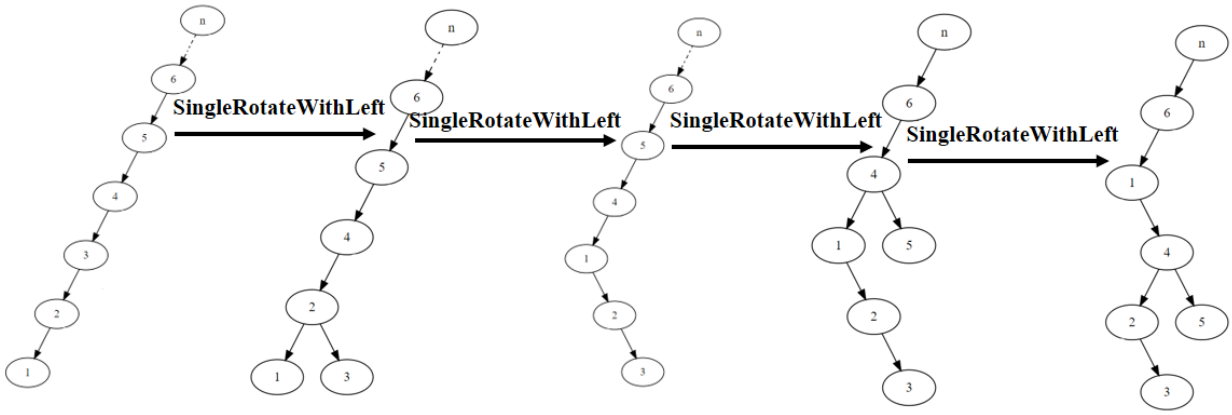
insert



In this case, it costs $O(1)$ to find the right place to place, and rotate the tree to make the inserting node be the root node. So the overall time complexity of insertion is:

$$\sum_{i=1}^{n}O(1)=O(N)$$

delete



As it can be seen from the above figure, after node 1 is adjusted to the root node, each even node has a right child node, and each odd node is a leaf node, and the original tree's height is reduced by half.So $Deletetime = O(depth)(find) + O(depth)(splay)$, after every search, the chain reduce to half of its orginal depth approximately. The depth varies from N to $N/2$, $N/4$, $N/8$....1 and even smaller since N is decreasing. So we may arrive at the following formular:

$$Time \leq O(N) + O(N) + O(N/2) + O(N/2) + O(N/4) + O(N/4) + \ldots + O(1)$$

$$Time \leq O(N) * (1 + 1 + 1/2 + 1/2 + 1/4 + 1/4 + \ldots + 1/N)$$

$$1 + \sum_{i=1}^{\infty} \frac{1}{2 \times i} = 2$$

$$Time \leq c \times O(N)$$

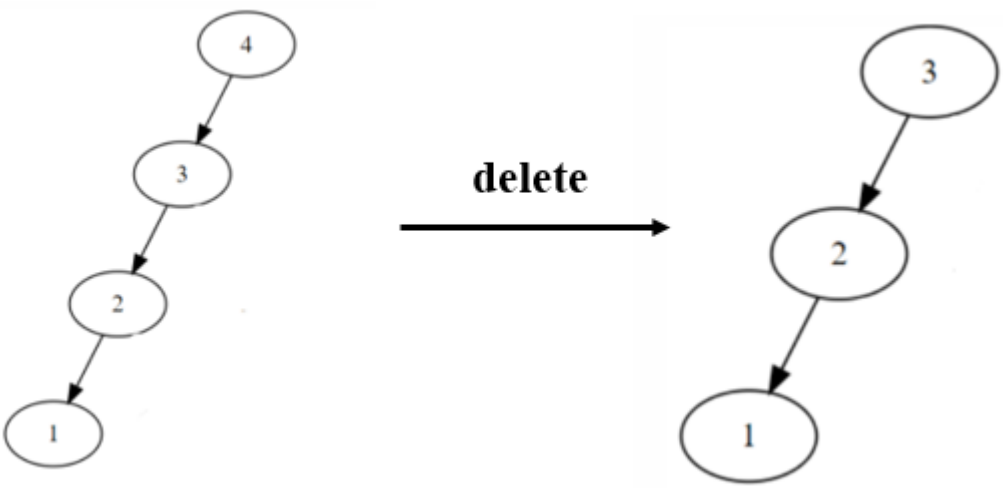
So the overall time complexity is $O(N)$

4.3.2 Insert N integers in increasing order and delete them in the reverse order

insert

The operation insert is same as above, we only talk about delete them in the reverse order.

delete



In this case, the node is always the root node. So it only costs $O(1)$ to delete the node.So the overall time complexity of insertion is:

$$\sum_{i=1}^{n} O(1) = O(N)$$

4.3.3 Insert N integers in random order and delete them in random order

background:

Amortized time efficiency of m operations is analyzed as following:

$$T_{amortized}(m) = O(m \log n)$$

It can be proved as following:

Define:

$size(r)$ = the Number Of Nodes in the sub-tree rooted at node r (including r).

$rank(r) = log_2(size(r))$.

$\Phi(T) = \sum_{i \in T} rank(i)$

$\hat{c_i} - c_i = Credit_i = \Phi(T_i) - \Phi(T_{i-1})$

zig:

$$\hat{c_i} = 1 + rank_2(X) - rank_1(X) + rank_2(P) - rank_1(P) \leq 1 + (rank_2(X) - rank_1(X))$$

zig-zag:

$$\hat{c_i} = 2 + rank_2(X) - rank_1(X) + rank_2(P) - rank_1(P) + rank_2(G) - rank_1(G) \leq 2 \times (rank_2(X) - rank_1(X))$$

zig-zig:

$$\hat{c_i} = 2 + rank_2(X) - rank_1(X) + rank_2(P) - rank_1(P) + rank_2(G) - rank_1(G) \leq 3 \times (rank_2(X) - rank_1(X))$$

It can be concluded that:

$$\hat{c_i} \leq 1 + 3 \times (rank_2(X) - rank_1(X))$$

However, there will be only once doing the operation of zig. So:

$$\frac{\sum c_i}{n} \leq \frac{\sum \hat{c_i}}{n} \leq 1 + 3 \times (R(T) - R(X))$$

Since that $R(T)$ is about $O(\log N)$,So the amortized cost for one operation is $O(\log N)$, and the total *amortized* time for a sequence of *m* operations is:

$$T_{amortized}(m) = O(m \log n)$$

insert analysis:

In this case, inserting N numbers randomly, so the time complexity is no more than $O(N \log N)$ .If we want to be more specific in this case, the overall time complexity is no more than:

$$\sum_{i=1}^{n} \log(i) = \log(N!) < N \log(N)$$

delete analysis:

When delete a node, it costs $O(\log N)$ to find the node, and then it costs $O(\log N)$ to Splay the tree to adjust the node to the root. After that, it need to adjust the maximum node in the left subtree to the root of the left subtree, which costs $O(\log N)$ .So:

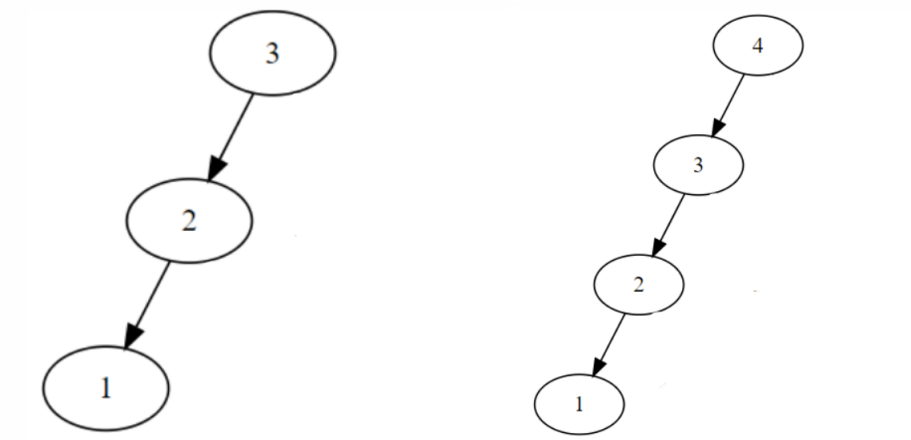
$$\sum_{i=1}^{n} \log(i) = \log(N!) < N \log(N)$$

As for the second delete method, the time complexity is the same as Unbalanced Binary Search Tree.

Algorithm 2: Top-down

4.3.1 Insert N integers in increasing order and delete them in the same order

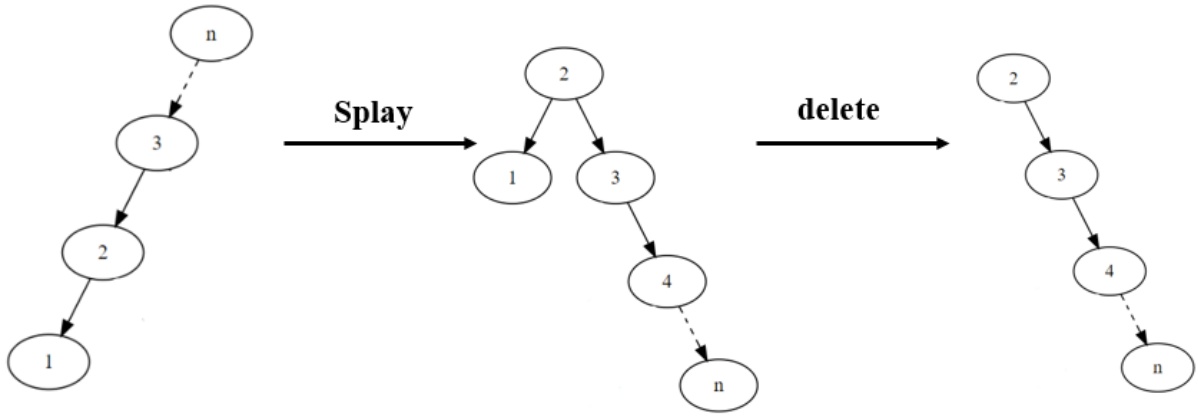
insert



In this case, inserting each node cost only $O(1)$ step. In the example as following, firstly, `Top_down_splay(4,T)` , and then this function only runs in $O(1)$ for that all the nodes in this subtree are smaller than 4. And then you only need to point the left pointer of node 4 to 3. So the total time complexity of insertion cost $O(N)$ .So the overall time complexity of insertion is:

$$\sum_{i=1}^{n} O(1) = O(N)$$

delete



In this case, it costs $O(N)$ to find the node with value 1 , and then it costs $O(N)$ to Splay the tree to adjust the node to the root. After that, it costs $O(1)$ to detete the node 1. Then the tree is what shown in the figure. Then it the node need to be deleted is always the root. So the overall time complexity of insertion is:

$$N + \sum_{i=1}^{N} 1 = O(N)$$

4.3.2 Insert N integers in increasing order and delete them in the reverse order

insert

The operation insert is same as above, we only talk about delete them in the reverse order.

delete

In this case, delete nodes in the same order. It costs $O(1)$ to find node $n-1$, and then run `Top_down_splay(n-1,T)` ,which costs $O(1)$ in this case. Finally, delete the root node in $O(1)$. So the overall time complexity of deleting N nodes is $O(N)$.

So the overall time complexity of insertion is:

$$\sum_{i=1}^{n} O(1) = O(N)$$

4.3.3 Insert N integers in random order and delete them in random order

Amortized time efficiency of m operations is analyzed as following:

$$T_{amortized}(m) = O(m \log n)$$

It has been proved in Bottom-up 4.3.3 .

insert analysis:

In this case, inserting N numbers randomly, so the time complexity is no more than $O(N \log N)$ .If we want to be more specific in this case, the overall time complexity is no more than:

$$\sum_{i=1}^{n} \log(i) = \log(N!) < N \log(N)$$

delete analysis:

When delete a node, firstly, run the function `Top_down_splay(X,T)` ,and the X node will be rightly at the root, it X node is exist, which costs about $O(\log N)$. And after that, it only costs $O(1)$ to delete the root node. So the overall time complexity is no more than $O(N \log N)$ .If we want to be more specific in this case, the overall time complexity is no more than:

$$\sum_{i=1}^{n} \log(i) = \log(N!) < N \log(N)$$

4.4 Conclusion

	Insert ↑	**Delete↑**	**Insert↑**	**Delete↓**
unbalanced BST	$O(N^2)$	$O(N)$	$O(N^2)$	$O(N^2)$
AVL Tree	$O(N \log N)$	$O(N \log N)$	$O(N \log N)$	$O(N \log N)$
Splay(Bottom-up)	$O(N)$	$O(N)$	$O(N)$	$O(N)$
Splay(Top-down)	$O(N)$	$O(N)$	$O(N)$	$O(N)$

	Insert(random)	**Delete(random)**
unbalanced BST	average:$O(N \log N)$	average:$O(N \log N)$
AVL Tree	$O(N \log N)$	$O(N \log N)$
Splay(Bottom-up)	$O(N \log N)$	$O(N \log N)$
Splay(Top-down)	$O(N \log N)$	$O(N \log N)$

Declaration

We hereby declare that all the work done in this project titled "Binary Search Trees" is of our independent effort as a group.

Reference

References:

[1] Introduction to Algorithms.

[2] Data Structure and Algorithm Analysis in C

[3] https://www.cnblogs.com/yangecnu/p/Introduce-Binary-Search-Tree.html

[4]https://en.wikipedia.org/wiki/Binary_tree

[5]https://en.wikipedia.org/wiki/Splay_tree

[6]https://en.wikipedia.org/wiki/AVL_tree