

Performance Comparison among Unbalanced Tree, AVL Tree and Splay Tree

Date: 2021-3-13

Chapter 1: Introduction

Binary trees are hierarchical data structures that allow insertion and a fast, nearest-neighbors search in one-dimensional data. Basic operations on a binary search tree take time proportional to the height of the tree. For a complete binary tree with n nodes, such operations run in $O(\log N)$ worst-case time. If the tree is a linear chain of N nodes, however, the same operations take $O(N)$ worst-case time.

To decrease the run time in the worst-case by changing the height of a binary tree, algorithms use rotation operations to create the AVL tree and the splay tree. The purpose of the project is to analyze and compare the performances of a sequence of insertions and deletions on unbalanced binary search trees, AVL trees, and splay trees.

According to the requirement, **the project tests the performance by implementing the following three sequences on the three different data structures:**

- (1) Insert N integers in increasing order and delete them in the same order.
- (2) Insert N integers in increasing order and delete them in the reverse order.
- (3) Insert N integers in random order and delete them in random order.

Besides, the distinct number N will be taken from 1000 to 10000, and the run times corresponding to the cases will be plotted with respect to the sizes to illustrate the difference.

The key algorithms to implement the trees are written in C, and the code to visualize the comparison among the three trees is written in python.

Chapter 2: Data Structure / Algorithm Specification

2.1 Unbalanced Binary Search Tree

An unbalanced binary search tree is simply a binary search tree(BST). It's called 'unbalanced' because **it could be unbalanced when the sequence of keys to be inserted is not so ideal**. A binary search tree satisfies that for any node X in the tree:

- (1) If a left subtree exists, then the values of all the keys in its left subtree are smaller than the key value in X .
- (2) If a right subtree exists, then the values of all the keys in its right subtree are larger than the key value in X .

Given its features, it is suitable for implementing binary-search and is commonly used for storing and accessing data. The definition of the node is as follow, which is the simplest among the three:

```

1. typedef struct Node *node;
2. struct Node
3. {
4.     int key;
5.     node left;
6.     node right;
7. };

```

Here, according to the demand, to compare its performance with the other two structures, **insertion** and **deletion** are needed in this project.

2.1.1 Insertion

When inserting a new key into the tree, we only need to **traverse the node in a way that following the features** of the tree. That is:

1. If the key is larger than the value of the current node X , we need to look for its position in the right subtree of X .
2. If no right subtree exists, then we create a node storing the key, which means the key has found its position in the tree;
3. If the key is smaller than the value of the current node X , the steps are similar to the two steps above.

By the steps above, we insert a key and keep the tree a binary search tree. The pseudo-code is as follow:

```

1. node Insert(int X, node T)
2.
3. if (!T)
4. {
5.     T = (node) malloc(sizeof(Node));
6.     T->Key = X;
7.     T->Left = T->Right = NULL;
8. }
9. else if (X < T->Key)
10.     T->Left = Insert(X, T->Left);
11. else if (X > T->Key)
12.     T->Right = Insert(X, T->Right);
13.
14. return T;

```

2.1.2 Deletion

When removing a node from a tree, we need to **find a suitable node to take its place**, or the tree will be taken apart. Assuming that we want to delete the

node containing the key X :

1. Find the node containing X by comparison.
2. If it has no subtrees, it can be easily deleted.
3. If it has only one subtree, the root of the subtree could take its place.
4. If it has two subtrees, the smallest key in its right subtree will be used to take the place of X , and then deletion is implemented again in the right subtree.

It's obvious that a function searching for the smallest key in a tree is needed in deletion. According to features, the function only needs to find the leftmost node in a tree. The pseudo-code is as follow:

```
1. node FindMin(node T)
2. {
3.     if (!T)
4.         return NULL;
5.     else if (!T->Left)
6.         return T;
7.     else
8.         return FindMin(T->Left);
9. }
```

Then, by the classification, the pseudo-code of deletion is as follow:

```
1. node Delete(ElementType X, SearchTree T)
2. {
3.     node Tmp;
4.
5.     if (!T)
6.         return NULL;
7.     else if (X < T->Key)
8.         T->Left = Delete(X, T->Left);
9.     else if (X > T->Key)
10.        T->Right = Delete(X, T->Right);
11.     else if (T->Left && T->Right)
12.     {
13.         Tmp = FindMin(T->Right);
14.         T->Key = Tmp->Key;
15.         T->Right = Delete(T->Key, T->Right);
16.     }
17.     else
18.     {
19.         Tmp = T;
20.         if (!T->Left)
21.             T = T->Right;
22.         else if (!T->Right)
23.             T = T->Left;
```

```

24.     free(Tmp);
25. }
26. }
27. return T;

```

In fact, lazy deletion can be adopted when the data size is small, but that is not what we want in the project.

2.2 AVL Tree

AVL trees are height-balanced binary search trees, which ensures the difference of the height of the left and the right sub-trees is less than 2. This difference is called the *Balance Factor*. According to the definition, the node in the AVL tree is defined as follow, in which *height* is added to calculate *Balance Factor*:

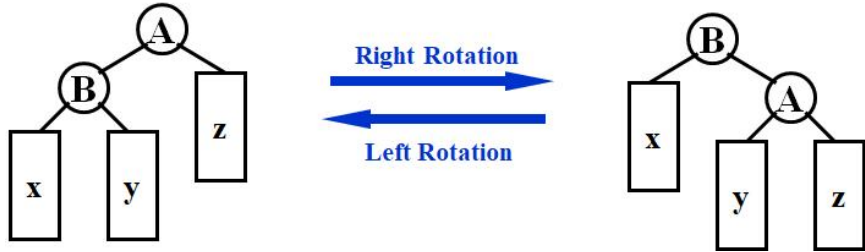
```

1. typedef struct node *node_ptr;
2. struct node
3. {
4.     int value;
5.     int height;
6.     node_ptr left_child;
7.     node_ptr right_child;
8. };

```

In the code, the height of an empty AVL node is defined to -1, and the height of one node with two NULL children is 0. The pointers (*left_child* and *right_child*) in the structure show the relationship between the node and its children. In the algorithm, we use top-down recursion to implement the operation, thus, the address of the parent node does not need to be saved in the data structure.

The key functions we used to adjust the height of the AVL tree’s nodes are the *left_rotation* and *right_rotation* functions. The following pseudo-code explain the method of rotation in detail:

Operation	
left_rotatio n pseudo-cod	<pre> 1. node_ptr left_rotation(node_ptr B) 2. { 3. new_root = B->right_child </pre>

e	<pre> 4. B->right_child = new_root->left_child 5. new_root->left_child = B 6. 7. B->height = 1 + Max(Height(B->left_child), Height(B->right_chi ld)) 8. new_root->height = 1 + Max(Height(new_root->left_child), Heigh t(new_root->right_child)) 9. 10. return new_root 11. }</pre>
right_rota tion pseudo-c ode	<pre> 1. node_ptr right_rotation(node_ptr A) 2. { 3. new_root = A->left_child 4. A->left_child = new_root->right_child 5. new_root->right_child = A 6. 7. A->height = 1 + Max(Height(A->left_child), Height(A->right_chi ld)) 8. new_root->height = 1 + Max(Height(new_root->left_child), Heigh t(new_root->right_child)) 9. 10. return new_root 11. }</pre>

There are four different conditions of a tree's structure when a tree or a subtree is not height-balanced. Based on the two basic rotations, we can use four types of rotation, namely LL, LR, RR, RL rotation, to adjust the position of nodes in the AVL tree.

Operation	Diagram	Pseudo-code
RR_rotation		<pre> 1. node_ptr 2. RR_rotation(node_ptr root) 3. { 4. return left_rotation(root); 5. }</pre>
RL_rotation		<pre> 1. node_ptr 2. RL_rotation(node_ptr root) 3. { 4. root->right_child = right_ro</pre>

		<pre> tation(root->right_child); 5. return left_rotation(root); 6. } </pre>
LL_rotation		<pre> 1. node_ptr 2. LL_rotation(node_ptr root) 3. { 4. return right_rotation(root); 5. } </pre>
LR_rotation		<pre> 1. node_ptr 2. LR_rotation(node_ptr root) 3. { 4. root->left_child = left_rota tion(root->left_child); 5. return right_rotation(root); 6. } </pre>

2.2.1 Insertion

The steps of insertion in an AVL tree are:

- (1) Firstly, insert the new node X as an unbalanced binary search tree.
- (2) From X to the root, update the height of all nodes in the path, and calculate the balance factor to check whether the tree or subtree is height-balanced.
- (3) If not, judge the type of the unbalance and implement rotations to regain balance.

The pseudo-code of this idea:

```

1. node_ptr Insert_AVL(ElementType value, node_ptr tree)
2. {
3.     if (tree is empty)
4.         create a one-node tree;
5.     else if (value < tree->value)
6.     { // insert to the left
7.         tree->left_child = Insert_AVL(value, tree->left_child);
8.
9.         if (left_height - right_height >= 2)
10.            if (value < tree->left_child->value)
11.                tree = LL_rotation(tree);
12.            else

```

```

13.         tree = LR_rotation(tree);
14.     }
15.     else if (value > tree->value)
16.     { // insert to the right
17.         tree->right_child = Insert_AVL(value, tree->right_child);
18.
19.         if (right_height - left_height >= 2)
20.         {
21.             if (value > tree->right_child->value)
22.                 tree = RR_rotation(tree);
23.             else
24.                 tree = RL_rotation(tree);
25.         }
26.     }
27.
28.     tree->height = 1 + Max(Height(tree->left_child), Height(tree->right_child))
29.     return tree; // new root
30. }

```

2.3.2 Deletion

The steps of deletion operation in an AVL tree are:

- (1) Firstly, find the node that needed to be deleted.
- (2) If two children both exist, then find the maximum node in the left subtree and copy the value to the node to be deleted, and delete the maximum node in the left subtree by recursion.
- (3) If one of the children is NULL, just return the pointer of the existed child.
- (4) If two children both are NULL, then free the buffer of that node. Such recursion ensures the node that being freed must be a leaf node, otherwise, the deleted node has only one child.
- (5) Lastly, do the same thing as the insertion, from the deleted node to the root, update the height of nodes in the path and rotate to regain balanced.

The pseudo-code of this idea:

```

1. node_ptr Delete_AVL(ElementType value, node_ptr tree)
2.
3.     if (tree->value == value)
4.     {
5.         if (both child exist)
6.             node_ptr max_in_left = FindMax_AVL(tree->left_child);

```



```

7.         tree->value = max_in_left->value; // use left max value to substitute the deleted node
8.         tree->left_child = Delete_AVL(max_in_left->value, tree->left_child);
9.     else
10.        // one of its child exist
11.        node_ptr temp_node = tree;
12.        if (tree->left_child == NULL)
13.            tree = tree->right_child;
14.        else if (tree->right_child == NULL)
15.            tree = tree->left_child;
16.        free(temp_node);
17.    }
18.    else if (tree->value > value)
19.        tree->left_child = Delete_AVL(value, tree->left_child); // the deleted node is in the left subtree of the node tree
20.    else if (tree->value < value)
21.        tree->right_child = Delete_AVL(value, tree->right_child); // the deleted node is in the right subtree of the node tree
22.
23.    if (tree != NULL)
24.    {
25.        if (getHeight(tree->left_child) - getHeight(tree->right_child) >= 2)
26.            if (getHeight(tree->left_child->left_child) > getHeight(tree->left_child->right_child))
27.                tree = LL_rotation(tree);
28.        else
29.            tree = LR_rotation(tree);
30.        if (getHeight(tree->right_child) - getHeight(tree->left_child) >= 2)
31.            if (getHeight(tree->right_child->left_child) < getHeight(tree->right_child->right_child))
32.                tree = RR_rotation(tree);
33.        else
34.            tree = RL_rotation(tree);
35.
36.        tree->height = 1 + Max(Height(tree->left_child), Height(tree->right_child));
37.    return tree; // new root

```

2.3 Splay Tree

The Splay tree is a kind of data structure derived from the binary search tree, which means it **inherits the features of a binary search tree**. Also, it's called

‘Splay’ because it has a special operation called ‘splaying’ to achieve self-balancing, which is also the most significant feature of the splay tree:

(1) Once a node is accessed, it is moved to the root by splaying, which actually is a series of rotations.

(2) Bottom-up splaying and top-down splaying are two ways to implement splaying.

Deletion is almost the same in the two methods, but insertion is not. For the **top-down** method, the tree is **adjusted until an empty root appropriate for the key appears**, then the key is put into the empty root. For the **bottom-up** method, the key is inserted into the splay tree like a binary search tree, then the node containing the newly inserted key is **moved up to the root**.

It is worth noting that, the structures of the node in the two methods are slightly different. For the **top-down** method, a node contains a pointer to its left child, a pointer to its right child and the key. For the **bottom-up** method, a pointer to the parent is needed also, which is shown as follow:

```
1. typedef struct Node *node;
2. struct Node{
3.     int key;
4.     node parent;
5.     node left;
6.     node right;
7.};
```

In the project, we adopted the bottom-up method.

Here, according to the demand, to compare its performance with the other two structures, insertion and deletion are needed in this project.

2.3.1 Splay & Find

It’s necessary to introduce splaying first. Splaying is actually a series of rotations, which is very similar to the rotation of AVL tree. The only difference is that, the single rotation for bottom-up method needs to **deal with the pointer to parent**. Take single right rotation for example:

```
1. node right_rotation(node T){
2.     node Left=T->left;
3.     node B=Left->right;
4.     node P=T->parent;
5.     Left->right=T;
6.     Left->parent=P;
7.     T->left=B;
8.     T->parent=Left;
9.     if(B)
10.         B->parent=T;
```

```

11. if(P)
12.     if(P->left==T)
13.         P->left=Left;
14.     else
15.         P->right=Left;
16. return Left;
17. }

```

Then, for splaying, according to what we learn in class, we need to discuss two cases for node X to be moved up:

- (1) If the node's parent is not the root and the node is not the root, implement a zig-zig or zig-zag rotation accordingly, which pushes the node up.
- (2) If the node's parent is the root, implement a single rotation.

The pseudo-code is as follow:

```

1. node Splay(node Tmp){
2.     while(Tmp!=root && Tmp->parent!=root)
3.     {
4.         if(L-L shape)
5.         {
6.             if(root == Tmp->parent->parent)
7.                 root=Tmp;
8.             right_rotation(Tmp->parent->parent);
9.             right_rotation(Tmp->parent);
10.        }
11.        else if(L-R shape)
12.        {
13.            if(root == Tmp->parent->parent)
14.                root=Tmp;
15.            LR_rotation(Tmp->parent->parent);
16.        }
17.        else if(R-L shape)
18.        {
19.            if(root == Tmp->parent->parent)
20.                root=Tmp;
21.            RL_rotation(Tmp->parent->parent);
22.        }
23.        else if(R-R shape)
24.        {
25.            if(root == Tmp->parent->parent)
26.                root=Tmp;
27.            left_rotation(Tmp->parent->parent);
28.            left_rotation(Tmp->parent);
29.        }

```

```

30.     }
31.     if(Tmp is the left child)
32.         return SingleRotationleft(root);
33.     else if(Tmp is the right child)
34.         return SingleRotationright(root);
35.     else
36.         return Tmp;
37. }

```

However, considering that every time a key is accessed, it's moved to the root. For insertion, if we find the key right after it is inserted, then it's moved to the root. For deletion, we just need to find the key to be deleted and then remove the root.

In short, we can **combine splay with search, insertion and deletion of a BST** to achieve the feature of the splay tree. The pseudo-code is as follow:

```

1. node Find(int key){
2.   node Tmp=BSTSearch(key, root);
3.   if(Tmp && Tmp!=root) // if the node exists, move it to the root
4.   {
5.       root = Splay(Tmp);
6.   }
7.   return root;
8. }

```

2.3.2 Insertion

Insertion is simple. The steps are:

- (1) Insert the new key like insert it into a binary search tree.
- (2) Splay the tree, moving the newly inserted key to the root.

With what we have written above, the pseudo-code is simple:

```

1. root=BSTInsert(key, NULL, root);
2. Find(key); // modify the tree after each insertion

```

The insertion here deals with pointer parent:

```

1. node BSTInsert(int key, node P, node T){
2.   if(!T) // the key has found its position
3.   {
4.       T=(node)malloc(sizeof(struct Node));
5.       T->key=key;
6.       T->left=T->right=NULL;
7.       T->parent=P;
8.   }
9.   else if(key < T->key)
10.  {
11.      T->left=BSTInsert(key, T, T->left);

```

```

12. }
13. else if(key > T->key)
14. {
15.     T->right=BSTInsert(key, T, T->right);
16. }
17. return T;
18. }

```

2.3.3 Deletion

Deletion is simple too:

- (1) Find the node X to be deleted and X will become the root.
- (2) Remove the root.
- (3) Find the node containing the largest key in the left subtree, which will become the new root, and make the right subtree the right child of the root.

The function to find the largest key is simple and its detail is left out. Here we give the pseudo-code of deletion:

```

1. node Delete(int key){
2. Find(key); //actually root->key == key
3. if(root->left) // even if the right child is NULL
4. {
5.     node Tmp=FindMax(root->left);
6.     root->key=Tmp->key;
7.     make Tmp the new root;
8.     free(Tmp);
9. }
10. else if(root->right) // root has only one right child
11. {
12.     root=root->right;
13.     free(root->parent);
14.     root->parent=NULL;
15. }
16. else // no children
17. {
18.     free(root);
19.     root=NULL;
20. }
21. return root;
22. }

```


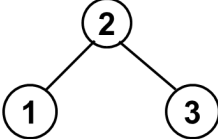
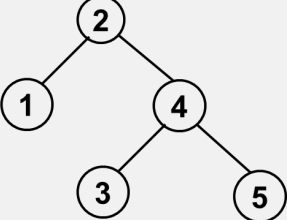
Chapter 3: Test

3.1 Correctness of Code

To check the results of operations, use a `levelorder_traversal` function to print the tree in level order. The pseudo-code of the level order traversal function is as follow:

```
1. void levelorder_traversal(node_ptr tree)
2. {
3.     enqueue(tree)
4.     while (queue is not empty) {
5.         visit ( T = dequeue ( ) );
6.         for(each child C of T)
7.             enqueue ( C );
8.     }
9. }
```

Considering the page, we do not show the detailed result of every function here. Take the insertion of an AVL tree for an example:

Cases/Inputs	tree	Results (level order traveler)	purpose
1 3		3	Create a one-node tree
3 3 2 1		2 1 3	Simple LL rotation
5 3 2 1 4 5		2 1 4 3 5	Simple RR rotation

10 3 2 1 4 5 6 7 16 15 14		4 2 7 1 3 6 15 5 14 16	RL rotation
15 3 2 1 4 5 6 7 16 15 14 13 12 11 10 8		7 4 13 2 6 11 15 1 3 5 10 12 14 16 8	No rotation
16 3 2 1 4 5 6 7 16 15 14 13 12 11 10 8 9		7 4 13 2 6 11 15 1 3 5 9 12 14 16 8 10	LR rotation

The conclusion is that the programs we write all function correctly.

3.2 Performance Test

First of all, we write a python program to generate random input series, which we input an integer N to will output three lines in the following format:

```
N a1 a2 a3 ...(ascending)... an a1 a2 a3 ..an
N a1 a2 a3 ...(ascending)... an an ... a3 a2 a1
N (random) (random)
```

Then the python program will output the time costs in the input series. We use a bash script(in Linux) to organize all the data.

Here, we focus on the bash script which generates the sample and use it to test the three trees. For convenience, we use .csv files to organize the result data.

1. To begin our test, we use the command `echo "insert_time,delete_time"` to initialize csv files.

```
#initiallize the csv files with the strings
echo "insert_time,delete_time" > ../time/Splayref_seq_time.csv
echo "insert_time,delete_time" > ../time/unbalancedref_seq_time.csv
echo "insert_time,delete_time" > ../time/AVL_seq_time.csv
echo "insert_time,delete_time" > ../time/AVL_anti_time.csv
```

2. Make use of a python program named *sample.py* to get the insertion sequence and the deletion sequence.

3. The sequences are together in the terminal. To make them into three files, we use the command `split -1 sample_init input_` to generate three input files, each corresponding to a case. Then we use `temp=cat input_aa | ../Splayref` to save them as files.

4. The sequence generated by python is separated by space. To make it a csv file, we need to replace the space by comma:

```
temp_1=${temp_1/Insert $i keys into the tree costs: }
temp_1=${temp_1/Delete keys costs:}
#organize the csv files
echo $temp_1|sed "s/ /,/g" >> ../time/Splayref_seq_time.csv
```

5. Repeat the steps above to get the sequences of all demanded sizes;

```
for i in {1000..10000}          #traverse the N from 1000 to 10000
do
    python sample.py $i > sample_init
    #get the input series which lines refer to three different input series
    split -1 sample_init input_
    #input the sequence insert series and sequence delete series
    temp_1=`cat input_aa | ../Splayref`
    temp_2=`cat input_aa | ../unbalancedref`
    temp_3=`cat input_aa | ../AVL_Tree`
    temp_1=${temp_1/Insert $i keys into the tree costs: }
    temp_1=${temp_1/Delete keys costs:}
    #organize the csv files
    echo $temp_1|sed "s/ /,/g" >> ../time/Splayref_seq_time.csv
    temp_2=${temp_2/Insert $i keys into the tree costs: }
    temp_2=${temp_2/Delete keys costs:}
    echo $temp_2|sed "s/ /,/g" >> ../time/unbalancedref_seq_time.csv
    temp_3=${temp_3/Insert $i keys into the tree costs: }
    temp_3=${temp_3/Delete keys costs:}
    echo $temp_3|sed "s/ /,/g" >> ../time/AVL_seq_time.csv

    temp_1=`cat input_ab | ../Splayref`
    temp_2=`cat input_ab | ../unbalancedref`
    temp_3=`cat input_ab | ../AVL_Tree`
    temp_1=${temp_1/Insert $i keys into the tree costs: }
    temp_1=${temp_1/Delete keys costs:}
    echo $temp_1|sed "s/ /,/g" >> ../time/Splayref_anti_time.csv
    temp_2=${temp_2/Insert $i keys into the tree costs: }
    temp_2=${temp_2/Delete keys costs:}
    echo $temp_2|sed "s/ /,/g" >> ../time/unbalancedref_anti_time.csv
    temp_3=${temp_3/Insert $i keys into the tree costs: }
    temp_3=${temp_3/Delete keys costs:}
    echo $temp_3|sed "s/ /,/g" >> ../time/AVL_anti_time.csv

    temp_1=`cat input_ac | ../Splayref`
    temp_2=`cat input_ac | ../unbalancedref`
    temp_3=`cat input_ac | ../AVL_Tree`
    temp_1=${temp_1/Insert $i keys into the tree costs: }
    temp_1=${temp_1/Delete keys costs:}
    echo $temp_1|sed "s/ /,/g" >> ../time/Splayref_random_time.csv
    temp_2=${temp_2/Insert $i keys into the tree costs: }
    temp_2=${temp_2/Delete keys costs:}
    echo $temp_2|sed "s/ /,/g" >> ../time/unbalancedref_random_time.csv
    temp_3=${temp_3/Insert $i keys into the tree costs: }
```



```

temp_3=${temp_3/Delete keys costs:}
echo $temp_3|sed "s/ /,/g" >>../time/AVL_random_time.csv
##print the process since the execute time will be over an hour
if [ $((($i % 100)) -eq 0) ];
then
    echo $i
fi
done

```

By far, we have obtained the data of running time, all of which is stored in .csv files. The total data is in the folder *time*. One example of size 10,000 is as follow:

Case	Time cost
Insert to AVL tree in increasing order	0.001990
Insert to AVL tree in random order	0.002877
Delete from AVL tree in increasing order	0.001736
Delete from AVL tree in decreasing order	0.002144
Delete from AVL tree in random order	0.003185
Insert to splay tree in increasing order	0.000467
Insert to splay tree in random order	0.006405
Delete from splay tree in increasing order	0.000865
Delete from splay tree in decreasing order	0.000168
Delete from splay tree in random order	0.004564
Insert to BST in increasing order	0.237819
Insert to BST in random order	0.001718
Delete from BST in increasing order	0.001678
Delete from BST in decreasing order	0.258702
Delete from BST in random order	0.000163

3.3 Visualization

We make use of matplotlib to visualize the result. The code is too much for the report, so we just show part of it here.

1. In the file *plot.py* which visualize the result, we first read the csv files storing the result data, like read in the running time of the three trees:

```

seq_AVL=pd.read_csv('../time/AVL_seq_time.csv')
seq_Splay=pd.read_csv('../time/Splayref_seq_time.csv')
seq_unbalanced=pd.read_csv('../time/unbalancedref_seq_time.csv')
...

```

2. For convenience, we need some reformed data frames, like the time for inserting and deleting in creasing order of the three trees:

```

seq_insert=pd.DataFrame({'AVL_insert':seq_AVL['insert_time'],
                        'Splay_insert':seq_Splay['insert_time'],
                        'unbalanced_insert':seq_unbalanced['insert_time']})
seq_delete=pd.DataFrame({'AVL_delete':seq_AVL['delete_time'],
                        'Splay_delete':seq_Splay['delete_time'],
                        'unbalanced_delete':seq_unbalanced['delete_time']})
...

```

3. Then we adjust the index of each data frame by adding 1000 to make it consistent with the actual size:

```
seq_insert.index=range(1000,10001,1)
seq_delete.index=range(1000,10001,1)
...
```

4. Here we are able to plot the result, with the following code:

```
plt.rcParams['figure.figsize'] = (32.0, 12.0) # 单位是 inches
plt.suptitle('TIME VS SIZE When Inserting and Deleting in Increasing
Order',fontsize=30)
xt=np.linspace(1000,10001,30)
ax1=plt.subplot(1,2,1)

# data_insert.plot()
#第一幅子图, 设置字体、图例、标题、坐标轴刻度
plt.plot(seq_insert['AVL_insert'],label='AVL_insert')
plt.plot(seq_insert['Splay_insert'],label='Splay_insert')
plt.plot(seq_insert['unbalanced_insert'],label='unbalanced_insert')

plt.title("TIME VS SIZE When Inserting in Increasing Order",fontsize=20)
plt.xlabel("N")
plt.ylabel("time cost")
plt.legend(loc="best",fontsize=15)
plt.xticks(xt)
ax2=plt.subplot(1,2,2)
# data_delete.plot()
#第二幅子图, 设置字体、图例、标题、坐标轴刻度
plt.plot(seq_delete['AVL_delete'],label='AVL_delete')
plt.plot(seq_delete['Splay_delete'],label='Splay_delete')
plt.plot(seq_delete['unbalanced_delete'],label='unbalanced_delete')

plt.title("TIME VS SIZE When Deleting in Increasing Order",fontsize=20)
plt.xlabel("N")
plt.ylabel("time cost")
plt.legend(loc="best",fontsize=15)
plt.xticks(xt)
plt.tight_layout()
#将图存成 png 文件

plt.savefig("../pictures/insert&seq_delete.png")
plt.show()
...
```

All the figures of the result will be shown in Chapter 4, or can be found in the folder *pictures*.

Chapter 4: Analysis

4.1 Complexities Analysis

Since the rotation operations only change the child pointer of a node, the space complexities of three binary trees are the same and proportional to the number of nodes, denoted N . Therefore, the space complexity of three binary trees is $O(N)$, the time complexities differ as follow.

4.1.1 Unbalanced Binary Search Tree

Firstly, we define *internal path length* as the sum of the depths of all nodes

in a tree, which is, for the root, **approximately the total number of comparisons to access all the data**. We also define $D(N)$ as the *internal path length* of a tree of N nodes. Clearly, we have:

$$D(1) = 0$$

By recurrence:

$$D(N) = D(i) + D(N-i-1) + N-1$$

In average cases, $D(i) = D(N-i-1) = \frac{1}{N} \sum_{j=0}^{N-1} D(j)$. Sum up the equation:

$$D(N) = \frac{2}{N} \sum_{j=0}^{N-1} D(j) + N-1$$

Thus the average time complexity is

$$\frac{D(N)}{N} = \frac{O(N \log N)}{N} = O(\log N)$$

However, that's not all. If the sequence of keys to be inserted is not so ideal, as a sequence in increasing order, **the tree becomes a chain**. In such case, it's easy to know the time complexity of accessing the data becomes:

$$\frac{D(N)}{N} = \frac{O(N^2)}{N} = O(N)$$

Also, the sequence of insertion and deletion influences the shape of the tree, which also influences the performance of the unbalanced binary search tree. That is, the unstable performance largely depends on the sequence of operations and the sequence of keys, which is mainly because the algorithm doesn't adjust the tree to keep it balanced and limits its application.

4.1.2 AVL Tree

Every rotation just change the child node of limited nodes, the number of assignment is fixed in rotation operation, so the time complexity of one rotation operation is **$O(1)$** . To insert a new node to an AVL tree with N nodes, the number of comparisons in the insertion function will no more than the height of the AVL tree. The AVL tree is height-balanced, so the time complexity of the insertion algorithm is **$O(\text{Height})$** .

Similarly, the time complexity of finding a distinct node in the AVL tree is **$O(\text{Height})$** . We know that the algorithm ensures the node being freed must be a leaf node, otherwise the deleted node just has one child node. In the worst case, the number of nodes in the path will not exceed the height of the tree, and every operation on these nodes is **$O(1)$** . Thus the time complexity of the deletion algorithm of an AVL tree is **$O(\text{Height})$** .

Let n_h denote that the minimum number of nodes in an AVL tree of height h .

$$n_h = n_{h-1} + n_{h-2} + 1$$

Compare to Fibonacci numbers:

$$F_0 = 0, F_1 = 1, F_i = F_{i-1} + F_{i-2} \text{ for } i > 1$$

We could find that:

$$n_h \approx \frac{1}{\sqrt{5}} \left(\frac{1 + \sqrt{5}}{2} \right)^{h+2} - 1$$

$$h = O \left(\log_{\frac{1+\sqrt{5}}{2}} n \right)$$

In conclusion, the time complexities of insertion and deletion operation are $O(\log N)$.

4.1.3 Splay Tree

Here we use amortized analysis to analyze an average case. Firstly we define D_i as the root of the resulting tree. We define the potential function as $\Phi(T) = \sum \log S(i)$, where $S(i)$ is the number of descendants of i (i included).

Discussing cases of zig, zig-zag and zig-zig, we know the amortized cost of each operation is

$$3(\Phi(T) - \Phi(X)) + 1 = O(\log N)$$

Where X is the new root and T is the original root. The time complexity in other cases is somehow too complex to analyze here, we only need to know that the worst time complexity can be

$$O(N)$$

By the way, it is necessary to analyze the characteristic of the splay tree.

- (1) Splay tree is created based on the belief that ***recently accessed elements are quick to access again***. That is, recently accessed data will be left on the top of a splay tree, which makes accessing to them quicker.
- (2) Every time a splaying happens, the tree is roughly halved, which benefits access to almost all the data.
- (3) Splay tree doesn't guarantee performance under the worst situation like when a non-decreasing sequence is inserted, the tree becomes a chain.

4.2 Sequence Operation

The specific running time may vary depending on the machine, compiler, and environment. We only need to pay attention to the **overall trend** here. Here we analyze the result, 6 figures, respectively. It's worth noting that the time cost shown in the figures is the **aggregate cost** of a series of operations, which may represent $O(N\log N)$ or $O(N^2)$.

The animation attached(in folder *gif*) can **help to understand the features we state below**.

Firstly we compare the performances when inserting and deleting both in increasing order, of which the figure is Fig 4.1.

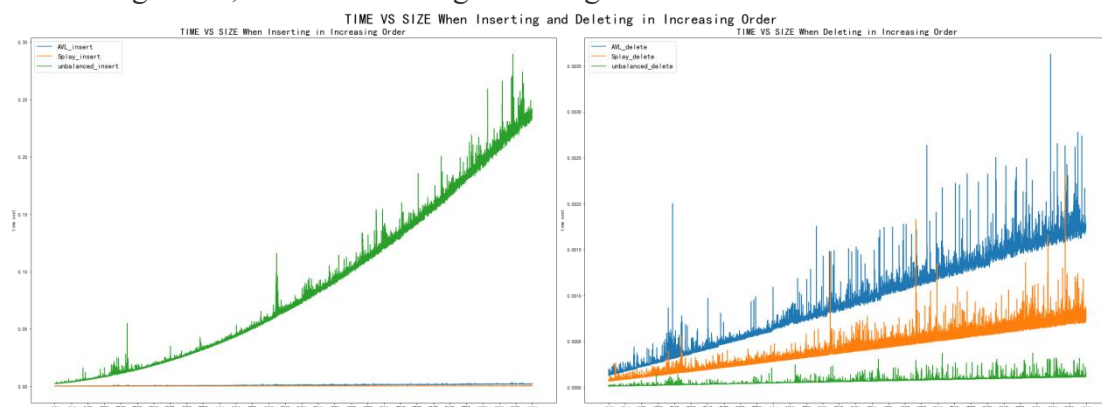


Fig 4.1 Inserting and Deleting in Increasing Order

When an increasing sequence is inserted, the performance of BST is absolutely more terrible than the other two. The reason is simple: inserting an increasing sequence into a BST is like creating a linked list, and the newly inserted node is always inserted to the end of the list. The BST here is **actually a chain**, which tremendously slows it down. In this condition, the time complexity of sequence insertion in BST is $O(N^2)$; by contrast, the AVL tree and the splay tree both are $O(N\log N)$.

When the keys are deleted in increasing order, the difference is obvious. For BST, deleting is actually **simply removing the root**, thus its performance is significantly excellent, with the time complexity of $O(1)$ in one deletion. For AVL tree, we postulate that **it's the frequent adjustment degrades its performance**. For the splay tree, it's **a chain too** after insertion. However, **the costly operations at the beginning decrease the cost of sequent deletion**, which proves its good amortized performance. In all, this case is **of great advantage to BST**, whose time complexity of sequence operation is $O(N)$, and the performances of splay tree and AVL tree are **slightly different but both good in general**, with the time complexity $O(N\log N)$.

Next, we compare the performances when inserting in increasing order and deleting in decreasing order, of which the figure is Fig 4.2. Since the performance of BST here shows the **difference of magnitude** compared to the

other two, we show the result in logarithm.

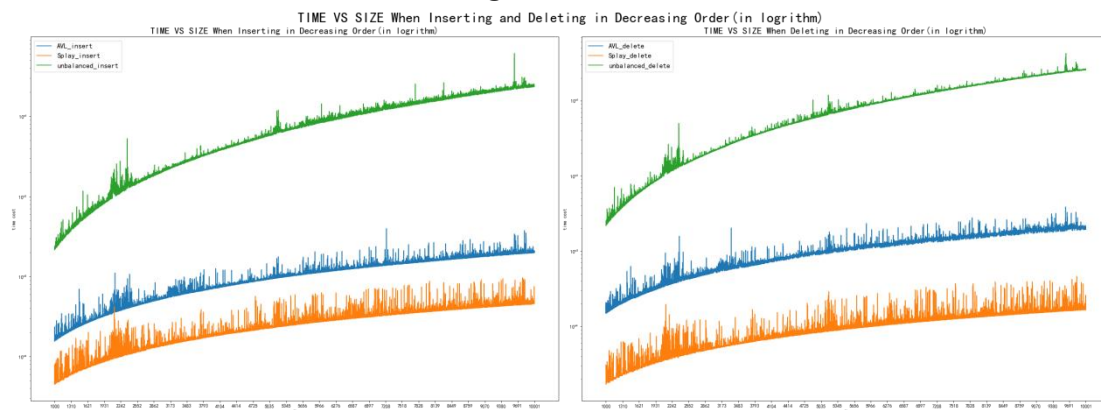


Fig 4.2 Inserting in Increasing Order and Deleting in Decreasing Order

The performance of insertion is almost the same as the one above. Because of the logarithm, here we can see clearly that, when inserting, the performance of an AVL tree is **slightly worse** than a splay tree (the difference is not that large as the figure shows, considering logarithm). We still attribute the difference to the frequent adjustment.

When the keys are deleted in decreasing order, the difference is significant. For BST, deleting is actually **removing the last node in a list every time**, which dramatically slows it down and proves its unstable performance. For the splay tree, this case is of advantage to it. Since **the lastly inserted (accessed) keys are left on the top of the tree**, they can be quickly accessed in a splay tree, which accelerates the performance. In all, this case is **of advantage to splay tree**, the performance of BST can be no more terrible and AVL tree are **stably efficient**.

Next we compare the performances when inserting and deleting both in random order, of which the figure is Fig 4.3.

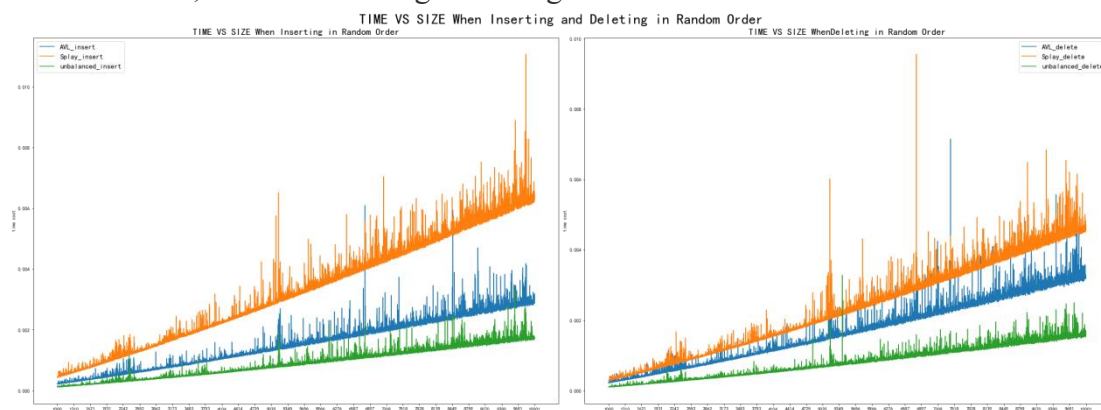


Fig 4.3 Inserting Deleting in Random Order

The case here is a typical average one. BST shows the best performance since it has **no adjustment** and the **random sequence helps it gain balance**. The AVL tree shows stable performance. Also, the randomness guarantees that **less adjustment happens**. The splay tree here shows the worst performance. We attribute it to mainly 2 reasons: one is that **adjustment of a splay tree is**

unavoidable in this case; one is that **the balance given by randomness may be ruined** by the adjustment that ‘roughly halves’ the tree.

However, it’s worth pointing out that, the difference is not that large and thus the three trees show similarly good performance in general in this case.

Finally, let’s see the performance of each tree, which is shown in Fig 4.4 and Fig 4.5.

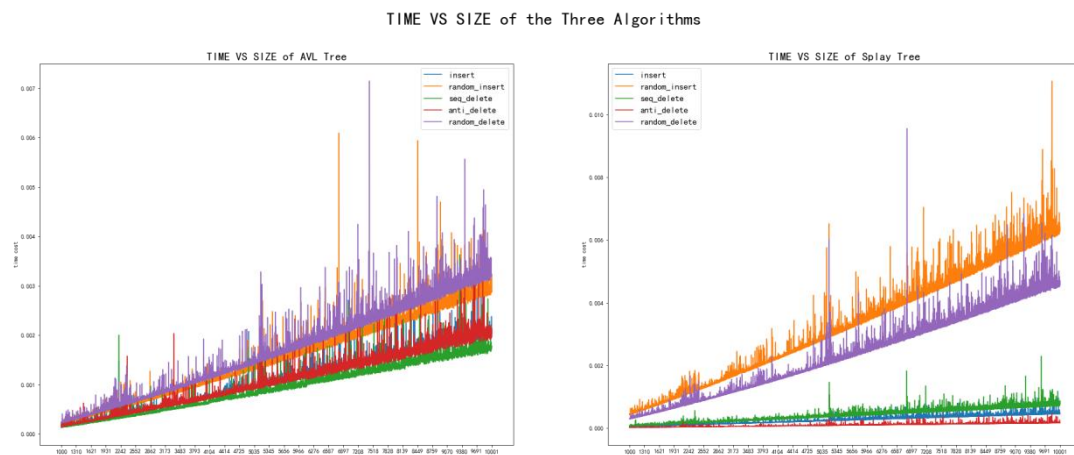


Fig 4.4 Performance of AVL tree and splay tree

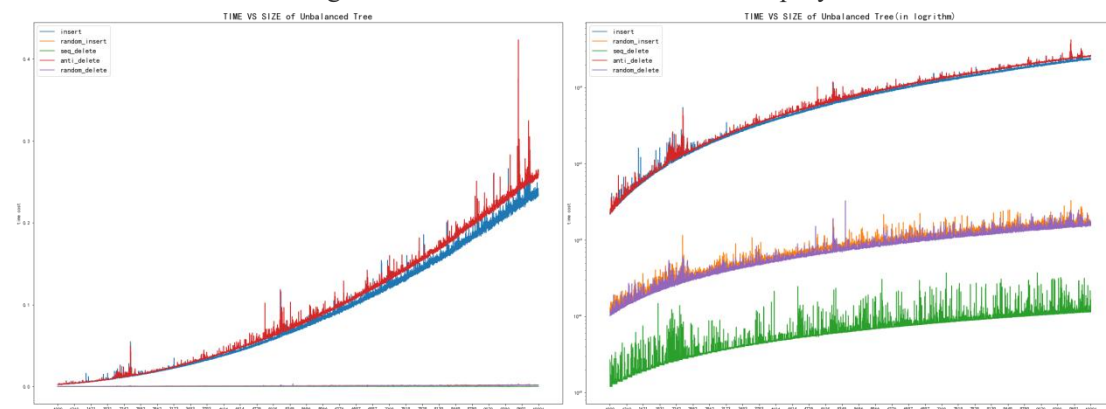


Fig 4.5 Performance of BST

It’s not hard to find out that the AVL tree is **the most stable one** among the three, owing to the balance gained by frequent adjustment.

Then, the splay tree is the second stable one. It **shows almost the same efficiency in average cases** but **shows bad performance $O(N)$ in worse cases**. In fact, the test in the project is somehow unfair to splay tree. The features of splay tree bring it 2 characteristics: one is that **the height of the tree may be reduced every time access happens, which benefits the access to all data**; one is that **recently accessed data is left on the top of the tree, which benefits the access to recently accessed data**. If the test includes a test for the running time of random access(visiting), allowing multiple access(visiting), we believe that splay tree will show better performance.

BST is the most unstable one. However, **in average cases, it’s the best one**.

Chapter 5: Conclusion

After the test and analysis above, we can conclude something.

For an unbalanced binary search tree:

1. Easy coding
2. Unstable performance: terrible performance under worst situation but excellent performance under average cases
3. No extra space is needed
4. Suitable for binary search in an average case
5. Show better performance when adjustment is introduced

For an AVL tree:

1. Difficult coding.
2. Very stable performance, $O(\log N)$ for every single operation.
3. Extra space is needed for bookkeeping data
4. Suitable for structures with features:
 - 1) Needing single operation mostly
 - 2) Little differences in the frequency of access to various data
 - 3) High demands on a single operation

For a splay tree:

1. Relatively easy coding.
2. Unstable performance: bad performance under worst situation but average-case performance is as efficient as other trees
3. No extra space is needed(for the top-down method)
4. Suitable for structures with features:
 - 1) Needing a series of operations or large-scale operation mostly
 - 2) Significant differences in the frequency of access to various data
 - 3) Moderate demands on a single operation
5. If a test for random access(visiting), allowing multiple access(visiting), a splay tree will show better performance than the other two.

Appendix: Source Code (if required)

The code is too much for the report. Please read the code in the attached files.

References

- [1] Mark Allen Weiss, "Data Structures and Algorithm Analysis in C, Second Edition", 机械工业出版社, 2019.11
- [2] "Splay tree", https://en.wikipedia.org/wiki/Splay_tree

Author List

Declaration

*We hereby declare that all the work done in this project titled
"Performance Comparison among Unbalanced Tree, AVL Tree
and Splay Tree" is of our independent effort as a group.*

Signatures

Each author must sign his/her name here.