Binary Search Trees

Author:漆翔宇　　ID:3170104557

Date：2018-3-10

Chapter 1: Introduction

Binary Search Tree (BST) is a more efficient kind of data structure to manage a series of data whose elements need to be modified frequently. By using the structure of tree,BST is expected to lower down the cost of it's searching operation in contrast with the liner structure,which is virtually O(max height of the tree) against O(n);

However,ordinary BST's efficiency is unstable,which in the worst situation could just be the same to the liner structure. To solve the problem of stability,balanced Binary search trees are created which perfectly enhance the efficiency of BST.

This report will give a sight into two kinds of balanced BST:AVL and Splay. We will analyze these two structures' principle and compare their efficiency with the ordinary BST.

Chapter 2: Balanced BST

Section 1: what is a Balanced BST?

A balanced BST tree satisfies the rules below:

1. Every node of the tree has at most 2 sons.
2. Every descendant of a node's left sub-tree has a lesser key than the node's and every descendant of a node's right sub-tree has a greater key than the node's.
3. For every node of the tree,the height of it's left sub-tree and the height of it's right sub-tree differ at most 1.

Section 2:Why a Balance BST is efficient?

Let f(i) : the min quantity of a balanced BST with the height of i.
Define: the height of an empty tree is -1.
It's easy to show that f(0)=1 and f(1)=2;
What's more,it's evident that f(n)=f(n-1)+f(n-2)+1,which means that the sequence f is like the Fibonacci sequence and the general formula has the form like f(n)=a*b^n+c.
Thus,we can see that in a Balanced BST with n nodes,the level of the quantity of the height is O(logn),which means the Time Complexity of accessing any node of the tree is just O(logn).

Section 3: AVL tree

In ordinary BST, the order we insert the data will finally determine the form of the tree which is not flexible enough. And it's not easy to find a superior order that can make the tree formed has a relatively lower height.

Using the method of rotating the vertex, AVL tree is flexible enough that no matter what the order the data is inserted, it can guarantee the BST is balanced.

As the proprietary operations of the AVL tree are just delete and insert,we just introduce these two operations.

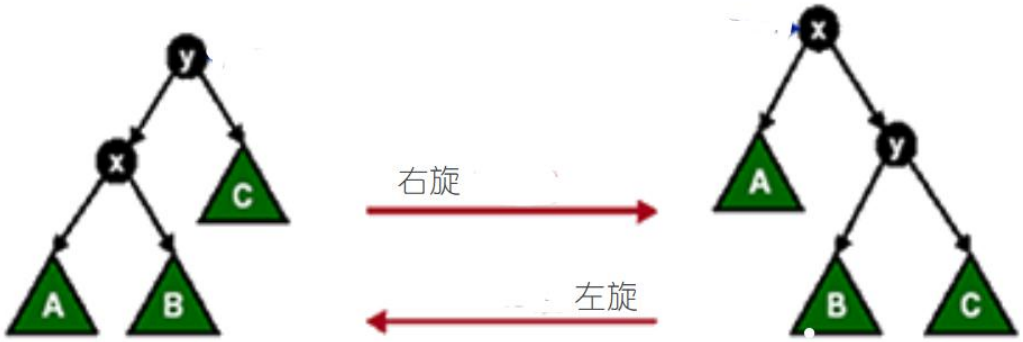
Part1: Some definition
For an AVL tree,the structure of a node is like below:

```
struct AVL
{
    int key,height;
    AVL *ls,*rs,*fz;
};
```

For a vertex t, the key is the parameter used to determine the position of the node. The height is just the height of the sub-tree with the root t(we will call it tree t below). Three pointers point the left son,the right son and the father respectively. Obviously,the tree t is balanced if and only if the height of it's left sub-tree and the height's of it's right sub-tree differ at most 1.

Part2: Rotation

AVL tree balances itself by rotating the vertexes,and any combination of rotations is based on just two kinds of rotation: turn left and turn right.(The are shower below.The picture is from the paper 《NOI2004 国家集训队论文》)



Take the right rotation for example,the pseudo-code is showed below:

```
right_rotation x
{
    y:x->fz; z:y->fz;
    if(y==z->ls)z->ls=x;else z->rs=x;
    x->fz=z;y->ls=x->rs;y->ls->fz=y;
    x->rs=y;y->fz=x;
    y->height=max(y->ls->height,y->rs->height)+1;
    x->height=max(x->ls->height,x->rs->height)+1;
    if(x->fz==_none)root=x;
}
```

Part 3: Insert

With Mathematical Induction,if we can construct a way of balancing the tree after inserting a new element in a originally balanced tree,the proposition that the tree we build is balanced is also proved.

Now,we define a parameter delta which is the difference between a node's left sub-tree height and right sub-tree height.

When we insert a new element into the AVL tree,we will starts from the root of the tree,and finally reach the position we insert the vertex. In this process,there will be a path in the tree from the root to the aim. After we have inserted the vertex,we just need trace back,updating the vertex's height along the path.

It is easy to show that the insertion operation's only influence to the balance of the tree is probably changing the height of some sub-trees thus changing the delta. And only the delta of the point in the path may change because of the insertion.

While tracing back,when we reach a vertex t,below is the all the situations we may meet:

1. Neither of the sub-tree of the vertex's height change. Respectively,the height of tree t

won’t change and of course the vertex’s ancestors’.So it’s no need to trace back anymore.

2. One of the sub-tree’s height changes(Obviously at most 1).

(i) Originally,the two sub-trees have the same height.

Then,the height of t will change ,and the delta changes to +1/-1。So we continue to trace back.

(ii) The former lower sub-tree’s height increases by 1.

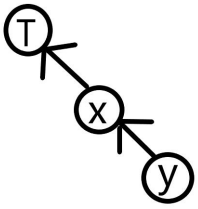
Then,the height of t remains unchanged and t is also balanced,stop tracing.

(iii) The former higher sub-tree’s height increases by 1.

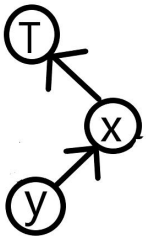
Then,the height of t increases by 1,and the sub-tree t needs to be shifted.

So,in the process of tracing back,the first time we meet an unbalanced sub-tree.we can concludes that: from the newly inserted node’s father to the now reached node’s son,all the vertexes have a delta with the value 1 or -1;

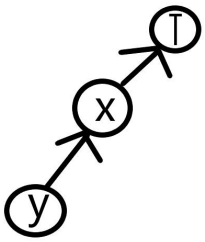
Now,we need to balance the tree by rotation.We have four kinds of compound rotation according to different tracing back paths.



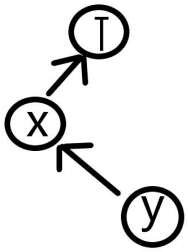
In this situation,just rotate_left(x)



In this situation,rotate_left(y) then rotate_right(y);



In this situation,rotate_right(x);



In this situation,rotate_left(y) and then rotate_right(y);

It’s easy to show,after the rotation operations above will guarantee the sub-tree

balanced again. And the height of the new sub-tree will be the same to what it is before inserting the new element which means after a rotation there is no need to trace back any more.

So in conclusion,when we insert a new element,we firstly access the position it is to be placed then trace back and update the height of the node we pass,until we meet a vertex unbalanced or root.

Now,we have already given a algorithm to insert element and keep balance. The time complexity of inserting is O(logn) of course.

Part 4:delete

When we want to delete an element,we first start from root to find the node to be deleted.

If the node has only one child,it is obvious that the only one sub-tree's height is 0. Delete the node and let it's only son replace it. Trace back to update the information,as the situation is symmetrical to the insertion,the algorithm is the same.

If the node has two child,choose the lower sub-tree(if the two sub-tree is equally high,either of them is acceptable.). If the chosen sub-tree is the left sub-tree,let the greatest node of it replace the deleted node. If the chosen sub-tree is the right sub-tree,let the least node of it replace the deleted node. Easy to proves that the height of the sub-tree won't change,and no need to trace back.

Section 4: Splay tree

Strictly,Splay tree is not a balanced BST. But as any consecutive m operations have a time complexity of m*logn which means the amortized time complexity of a single operation is logn,we can consider it as a special kind of Balanced BST.

As the PPT of our lesson has given the amortized analysis,here we won't talk about the time complexity any more. In this section,we mainly introduce how to implement the Splay tree.

As the splay operation,insert and delete are the proprietary operations of Splay tree,we just mention these three operations.

Part 1:Rotation

Splay tree is also based on the rotation operations.The two basic rotation method is the same to AVL. Here,we don't repeat it again.

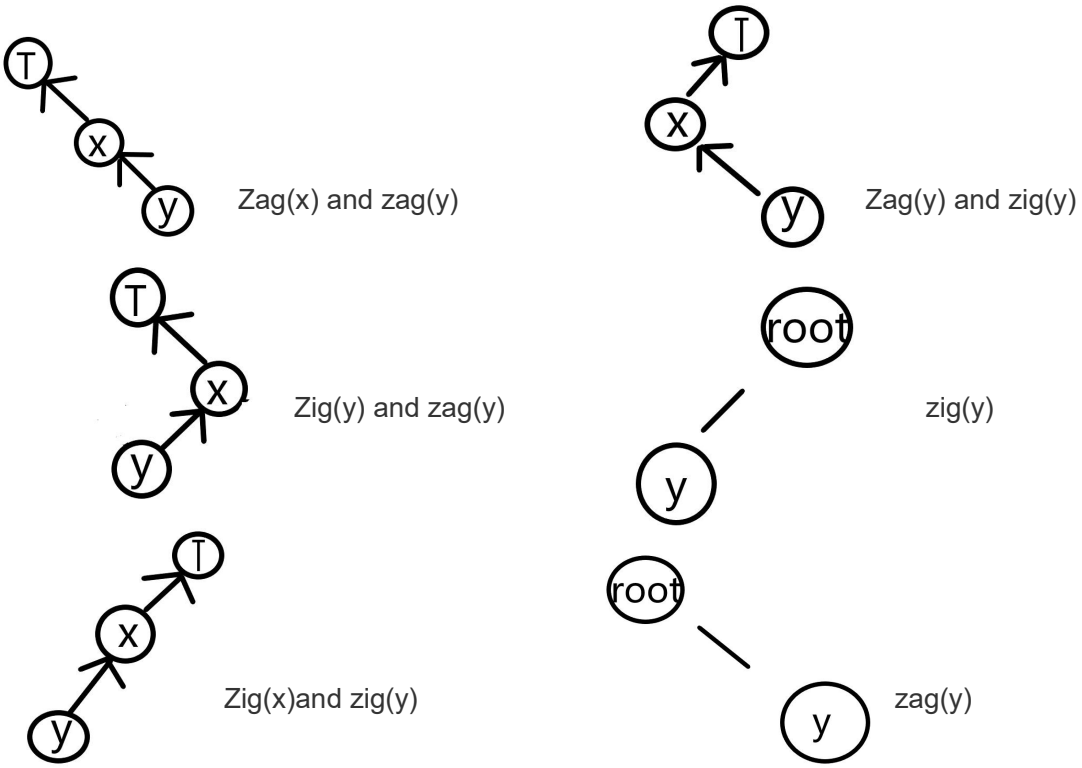
Part 2:Splay operation

Splay(x) is the core operation of the Splay tree. After splay(x) the node x will be the new root of the Splay tree.

How does splay(x) make it? By rotation!

In Splay tree,we call left_rotation in the AVL tree zag and call right_rotaion in the AVL tree zig.

When we spaly(x),six situations will be possible.



Using the six compound rotation until the y becomes the new root of the tree.

Part 3:Insert

Starts from root and find the position we insert the element. After inserting it,splay(the new node).

Part 4:delete

In fact,we can directly use the same delete method of AVL tree. But a more concise style is :

T=the one to be deleted

splay(T)

If T has no left son,delete T and make it's right son the new root.

Else

delete T

make it's left son the new root

splay(the max element of the tree)

Make T's right son the root's right son

Chapter 3: Testing results

In this chapter,we will compare the time efficiency among ordinary BST, AVL tree and Splay tree by doing experiment.

By running a data-maker,we have created 12 types of data,and each type we have 1000 groups,and each number of a single group is extinct. For each type of data,the MAX running time ,MIN running time and the average running time of a single group is recorded.

Section 1: The table of cases:

	Quantity of groups	Size of the n	Inserting order	Deleting order	Range of the integers
Case1	1000	1000	Increasing	Increasing	0~9999999
Case2	1000	1000	Increasing	Decreasing	0~9999999
Case3	1000	1000	Random	Random	0~9999999
Case4	1000	3000	Increasing	Increasing	0~9999999
Case5	1000	3000	Increasing	Decreasing	0~9999999
Case6	1000	3000	Random	Random	0~9999999
Case7	1000	7000	Increasing	Increasing	0~9999999
Case8	1000	7000	Increasing	Decreasing	0~9999999
Case9	1000	7000	Random	Random	0~9999999
Case10	1000	10000	Increasing	Increasing	0~9999999
Case11	1000	10000	Increasing	Decreasing	0~9999999
Case12	1000	10000	Random	Random	0~9999999

Section 2: Introduction and analysis to each case

The 12 cases can be divided into 3 classed or 4 levels.

1.The case with the same inserting and deleting order can be seen as of the same class.

(i) Increasingly insert and increasingly delete.

a.For the ordinary BST:
dealing with this class of cases cost O(n^2) while inserting and O(1) while deleting.
In total:O(n^2)+O(n);

b.For the Splay tree:
As every time we insert a new element,we will splay it. So as we insert an element, it only take O(1).And after all the insertion,the splay tree is a liner structure.
And when we start deleting,it will take O(n) for the first time. After that every time we delete,just take O(1);
So,dealing this class of data with Splay only take O(n)!!!
In total:about O(n)+O(n)+O(n);

c.For the AVL tree:
AVL tree is much more stable! Any time it is a strictly balanced tree,so any insertion or

deletion is expected O(logn);
In total:about O(n*logn)+O(n*logn)

(ii)increasingly insert and decreasingly delete

a. For the ordinary BST
The insertion and deletion are both O(n^2);
In total:O(n^2)+O(n^2)

b. For the Splay tree
The deletion is faster than that in class 1.
In total bout:O(n)+O(n)

c. For The AVL tree
In total about:O(n*logn)+O(n*logn)

(iii)randomly insert and randomly delete

a. For the ordinary BST:
It may be far more efficient than the two classes above.
We can expected a O(nlogn) result

b. For the Splay tree
It may take more time than that in the two classes above.
But we can guarantee the amortized time complexity is O(n*logn)

c. For the AVL tree
Should still be O(n*logn)

2.The cases with the same size of n can be seen as of the same level.
By larger the size of n,we can make the difference more obvious.

Section 3: The expected result and the actual the result.

In light of the analysis in section 2,we can have some qualitative prediction as follows:

	Class 1	Class 2	Class 3
Splay	x	(2/3)*x	C (C>x)
AVL	t (t>x)	t	t
BST	y (y>>x,y>>t)	2*y	Y (Y<y;Y>C;Y>t)

After the test,the actual result is as follows:

Splay：

```
case1: MIN=0 MAX=1 average=0.151
case2: MIN=0 MAX=1 average=0.092
case3: MIN=0 MAX=3 average=0.9
case4: MIN=0 MAX=2 average=0.521
case5: MIN=0 MAX=2 average=0.307
case6: MIN=2 MAX=7 average=3.296
case7: MIN=0 MAX=4 average=1.112
case8: MIN=0 MAX=2 average=0.682
case9: MIN=7 MAX=15 average=8.86
case10: MIN=0 MAX=4 average=1.654
case11: MIN=0 MAX=4 average=1.028
case12: MIN=11 MAX=23 average=13.418
```

AVL:

```
case1: MIN=0 MAX=1 average=0.343
case2: MIN=0 MAX=1 average=0.336
case3: MIN=0 MAX=1 average=0.461
case4: MIN=0 MAX=3 average=1.157
case5: MIN=0 MAX=5 average=1.123
case6: MIN=1 MAX=3 average=1.499
case7: MIN=2 MAX=5 average=2.655
case8: MIN=2 MAX=5 average=2.625
case9: MIN=3 MAX=6 average=3.953
case10: MIN=3 MAX=6 average=3.833
case11: MIN=3 MAX=6 average=3.84
case12: MIN=5 MAX=8 average=6.034
```

BST:

```
case1: MIN=3 MAX=6 average=3.967
case2: MIN=6 MAX=11 average=6.725
case3: MIN=0 MAX=3 average=1.576
case4: MIN=30 MAX=41 average=33.567
case5: MIN=58 MAX=76 average=63.899
case6: MIN=4 MAX=8 average=4.932
case7: MIN=204 MAX=262 average=216.893
case8: MIN=387 MAX=431 average=405.378
case9: MIN=10 MAX=20 average=11.959
case10: MIN=423 MAX=549 average=445.125
case11: MIN=806 MAX=889 average=843.687
case12: MIN=16 MAX=26 average=17.618
```

Time(ms)	Splay	AVL	BST
Case1	0.151	0.343	3.967
Case2	0.092	0.336	6.725
Case3	0.9	0.461	1.576
Case4	0.512	1.157	33.567
Case5	0.307	1.123	63.899
Case5	3.296	1.499	4.932
Case7	1.112	2.655	216.893
Case8	0.682	2.625	405.378
Case9	8.86	3.953	11.959
Case10	1.654	3.833	445.125
Case11	1.028	3.84	843.687
Case12	13.418	6.034	17.618

Tips:The cases with the same color are of the same class.

According to the experiment result,we can see that the predictions made before is right. But there are some other important information worth considering:

1.In the AVL tree and Splay tree,dealing with the the random data takes more time than the sorted data.

2.The AVL tree's efficiency is very stable! Dealing with the random data with AVL tree is faster than the Splay tree!

3.Although BST will be very slow in the worst situation,dealing with random data with BST is not such a bad choice. In fact,it's expected complexity is good.(Just a little worse than the Splay's amortized complexity.)

Chapter 4: Analysis and comments

It's easy to show that the space complexity among these three structures is basically the same.

As for the time complexity,as we have analyzed above,the AVL seems faster and more stable than the other two structures when dealing with the ordinary random data.

Just according to the general efficiency,it seems that AVL is the best choice.However, from my perspectives,the Splay tree is better. Yes,it is no much better than the BST when dealing with random data! But it is much easier to implement meanwhile it can guarantee the efficiency even in the worst situations ! And what's more? The structure of the Splay is much more flexible! You can make any node be the root. Neither the AVL nor the BST can not empower such flexibility!

Apart from these three structures,Treap is also a good choice.It is based on BST and heap!As we know,with the random algorithm,the BST can get a good expected time complexity. Treap is also using the thought of random algorithm. In a treap,the nodes not only have a key for the BST order,but also have a random parameter for the heap order. In other words,the Treap is not only a BST but also a heap! Treap is relatively harder to get into trouble(the worst situation) for that it is a good replacement for balanced BST.The biggest forte is that it's much more easy to implement. Limited to the pages,we won't talk about the detail of it.

Appendix: Source code

Please find them in the folder sent with the report.

References

[1] 杨思雨，《伸展树的基本基本操作与应用》, 2004NOI 国家集训队论文。
[2] Thomas H.Cormen,Chareles E.Leiserson,Ronadl L.Rivest,Clifford Stein,《Introduction to algorithm》Page 161~169.

Author list

漆翔宇 has done all the jobs.

Declaration

Hereby I declared that all the work done in this project titled “Binary Search Trees” is of my independent effort. The work is done as a **bonus.**