Comparing Different Binary Search Trees

group 8: 颜晗，吴俊贤，钱于飞

Abstract: To begin with,we implemented the insertion,deletion, query and height calculation of the ordinary binary search tree,the AVL tree ,the red-black tree and the splay tree. Second,we tested and compared their performance under different sequence. Finally we analyzed the tested result and got the conclusion in this project.

1. Description of the project

In this project , we are demand to implemented the ordinary binary search tree , the AVL tree, the red-black tree,and the splay tree. And then we need to compared their performance under different sequences and operations.

2. The theoretical performance

2.1 the ordinary binary search tree

```
typedef struct BSTNode{
    int value;
    int height;  //The height of the node
    struct BSTNode* left;
    struct BSTNode* right;
}BST_Node;
```

We use value to store the element of BST node and height to store the height of the node to help us calculate the height of the tree. And two pointers store the two children of the node.

For the ordinary binary search tree, the time complexity of each search, insertion and deletion is O(h),and h is the height of the tree. But since it does not adjust the resulting tree, in a good case the height might be $logN$,in some special cases,the tree is a straight line that the height will equal to N. However, we can prove that The height expectation of a randomly constructed binary search tree is O($logN$). That's say, it doesn't do too badly in general.

2.2 the AVL tree

```
typedef struct AVLTreeNode{
    int value;
    int height;  //The height of the node
    struct AVLTreeNode* left;
    struct AVLTreeNode* right;
}AVLTree_Node;
```

We use value to store the element of AVL tree node and height to store the height of the node to help us calculate the height of the tree and the balanced factor of all nodes. And two pointers store the two children of the node.

For the AVL tree,though it's a BST whose time complexity is O(h). As AVL tree is balanced, we can prove mathematically that the upper bound of its height is about $1.44 * log(N+2)$, which means the time complexity of insert, query, delete operation are all O($logN$).

2.3 the red and black tree

```
struct RBnode{
    int num;
    int color;
    RBptr left;
    RBptr right;
};
```

The data structure of red-black tree node contains two variables and two pointers: " num " stores the value of the node; "color" stores the color of the node, where 1 indicates red, 2 indicates black and 3 indicates double black; left and right are pointers pointing to left son and right son.

Because a red and black tree has some good properties, we can prove that the height of a red and black tree is at most $2log(N+1)$. It means the time complexity of its operation are also O($logN$).

2.4 the splay tree

```
typedef struct splaynode{
    int value;
    splayP left;
    splayP right;
    splayP parent;
} splaynode;
```

The "value" will stores the key of the node. Two pointer "left" and "right" will store a node's two son, the "parent" pointer will store its parent so that the splay operation can be easy.

As we learned in class, we could use the amortized analysis to find the time

complexity of the extended tree's operations. The amortized cost of all operations (even on different conditions) on the splay tree is O($logN$).

As we can see, the time complexity of their performance can all be O($logN$), only the BST seems to be a little weak. But the real performance need our test.

3. Implementation details

3.1 the ordinary binary search tree

Query:

According to the property of the BST, We used iterative way to design the search function. we just need to compare what we need to find with the key of a node ,then decide to go left or right until we find it.(In fact, the others trees also use this way ,except the splay tree)

Insertion:

If the new number is less than the root,insert it into its left tree,otherwise insert it into its right tree. Then we can go on the loop until encountering an empty node. Finally we can create a one-node tree jump out of the loop.

Deletion:

If the node that is to be deleted has no children,delete it directly. If the node that is to be deleted has only one child,we just need to connect the child to his parent. If the node that is to be deleted has two children,we can select a minimum number in the right subtree or a maximum number in the left subtree,which we can compare the height of subtrees to determine which number we will choose in order to improve the efficiency. So we can replace the deleted node with it in previous position of the deleted node, and delete the number in its subtree.

3.2 the AVL tree

Query:

Because an AVL tree is a BST, similarly , We used iterative way to design the search function.

Insertion:

In addition to the similar part to ordinary BST,for AVL tree,we introduce BF(node) = the absolute value of the difference between the left and right subtree heights. When the BF values of all nodes are 0 or 1,the tree is balanced. Therefore,every time we insert a tree, we need to analyze the BF of each node on the path from the node to the root from bottom to top to find the first element that makes the tree unbalanced. Then we using the location relationship between the inserted node and the problem node to divide the all cases into four kinds: LL, LR, RR, RL. For LL case and RR case,we just need to do single rotation. And for LR case and RL case,we need to do double rotation to make the tree balanced.

Deletion:

For the deletion of AVL tree,we used the same method as the ordinary BST to delete the node. But it should be noted that we should determine whether the tree is unbalanced or not. If after the deletion the tree becomes unbalanced,we need to consider each subtree containing the node from bottom to up. Similar to inserting a node in AVL tree,we can use BF and four kinds of unbalanced cases to adjust the tree imbalances.

3.3 the red and black tree

Query: same as AVL.

Insertion: Pseudo code.

```
Find the location for insertion,store the trail, mark the new node as red.
While(there is violation)
    If(father is a left node)
        If(uncle is red)
            Mark son and uncle black, mark father red.
            If(uncle if black)
                If(son is a right node)
                    Left rotate son
              If(son is a left node)
                    Right rotate father
       Else
        Contrary to “father is left node” case.
```

The restoration of tree is divided into two cases, father of the violation node is a left node or a right node. In each case, there are three minor cases: uncle is red(case 1), uncle is black while son is “zig-zig”(case 2.1), and uncle is black while son is “zig-zag”(case 2.2). Because case 1 can transform into case 2.1 and 2.2, case 2.2 can transfer into case 2.1, so the implement order is case1, case2.2, case 2.1. We use iteration and store the trail from root to inserted node in a pointer array, instead of using recursion, this may save time.

Deletion: Pseudo code.

```
Find the node to be deleted, store the trail.
If the node has two child, switch it with the largest node in its right subtree.
If the node is red, delete directly, done.
If the child of node is red, mark the child black and link it with the node’s father,done.
Mark the node “double black”.
While(current node is double black)
        If (node is a left node)
```

```
            If(sibling of node is red)
                Modify color
                Left rotate sibling
        If(sibling of node is black)
                If(far nephew is red)
                    Modify color
                    Left rotate sibling
                Else if(close nephew is red)
                Modify color
                    Double rotation
                Else  //both nephews are black
                    Modify color
                    If father is red or is root,done
                    Else, violation is propagated upwards
        Else
            Contrary to “node is a left node” case
```

There are two cases without violation: deleted node is red and deleted node has a red child. Other than the special cases, there are four cases that includes a “double black” node. Because of the transformation relationship of these cases, we check “sibling is red” case first and “sibling is black” case after.

3.4 the splay tree

Query:

Because when we find a key in splay tree, we need to splay it to root and when we do splay from bottle to top , we need to find the key. So we just use the splay function as query function. And for splay operation, there are five cases.

> 1. it's root.
> 2. its' root's son, single rotation.(left or right)
> 3. zig_zig, double rotation.(left or right)
> 4. zig_zag, double rotation.(left or right)
> 5. not found, just choose the last one as the new root.
>
> **Insertion:**
>
> First, we need to insert the key to tree as BST. Then we use splay function. As we need to do two things, we have two functions to do them. The big one need to insert to a NULL tree and do splay after insert, the small one just do insert.

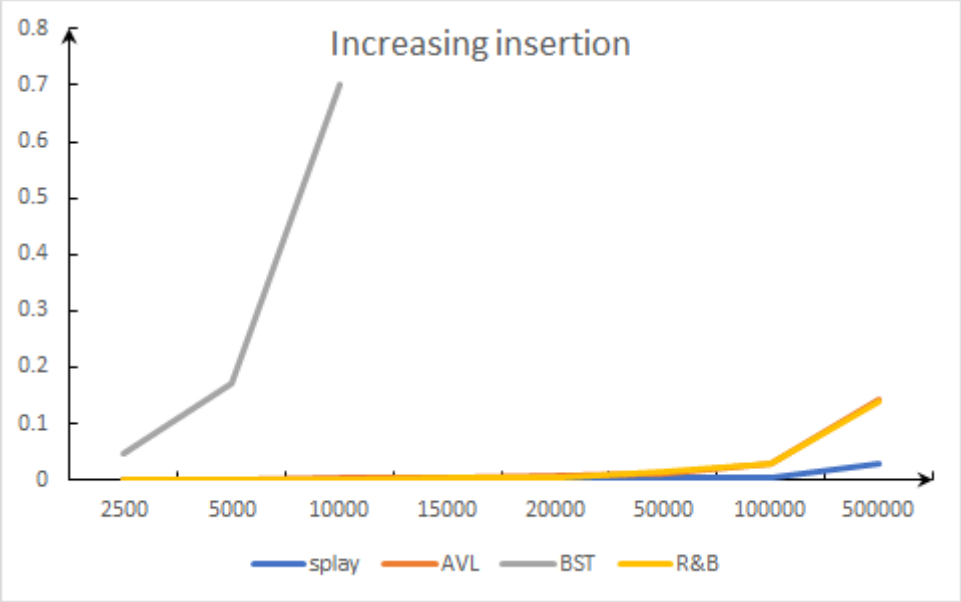
Deletion:

when we do deletion, we usually need to do three things. First, splay the key to root and delete it. Then we need to find the biggest key in left tree(If it exists. If not, just return the right one). Finally, we need to splay the biggest key of left tree to root and connect right tree to it.

4. Tests and analysis

What we show are just parts of the possible result, and the results are influenced by many factors when program runs in PC, so we only analyze these results qualitatively.

Increasing insertion:

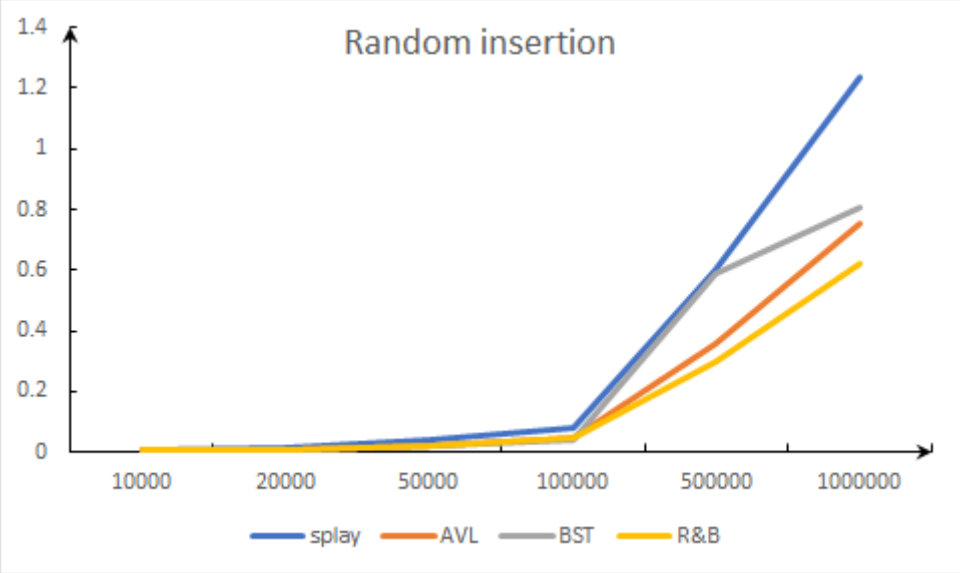


data/height	splay	AVL	BST	R&B
2500	no data	no data	2500	no data
5000	no data	no data	5000	no data
10000	10000	14	10000	24
15000	15000	14	15000	25
20000	20000	15	20000	26
50000	50000	16	no data	29
100000	100000	17	no data	31
500000	500000	19	no data	35

In this test, we will insert 1~N to these trees. As we can see, the BST cost the largest time, AVL tree and R&B tree cost almost the same time, and splay tree cost the least time. Because we're inserting in ascending order, the BST will be a line which all the node only has right son which means we need to do recursion many times after we insert many nodes, the time is much larger than others of course. AVL tree and R&B tree could balance the tree when inserting, and adjustment wouldn't cost too much, so the time cost is much smaller. The splay tree will put the new node in root, so its node only has left son after

inserting, and we just insert new key into the root's right son and splay it to root, it cost a few time that we almost can't measure it.

Random insertion:

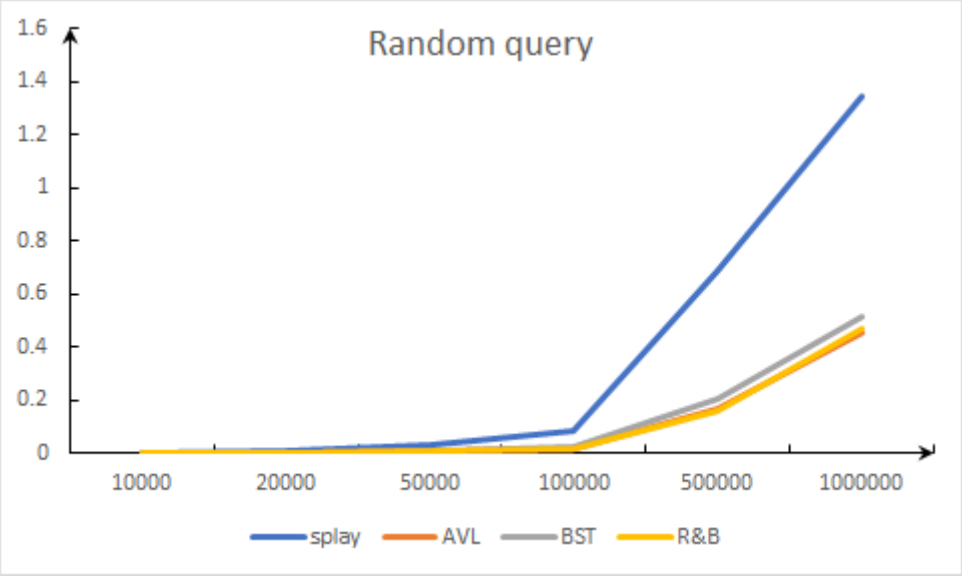


data/height	splay	AVL	BST	R&B
10000	39	16	29	16
20000	36	17	33	18
50000	41	18	35	19
100000	45	20	41	20
500000	59	23	47	23
1000000	54	24	54	24

How we get the random number: We will use the excel to do it. First, we generate numbers from 1 to N, then generate N random number besides. The excel can sort the random number and extend to ordered sequence, so the random sequence will be ordered and the ordered sequence will be random. Then we get a sequence in random order and no repeated number.

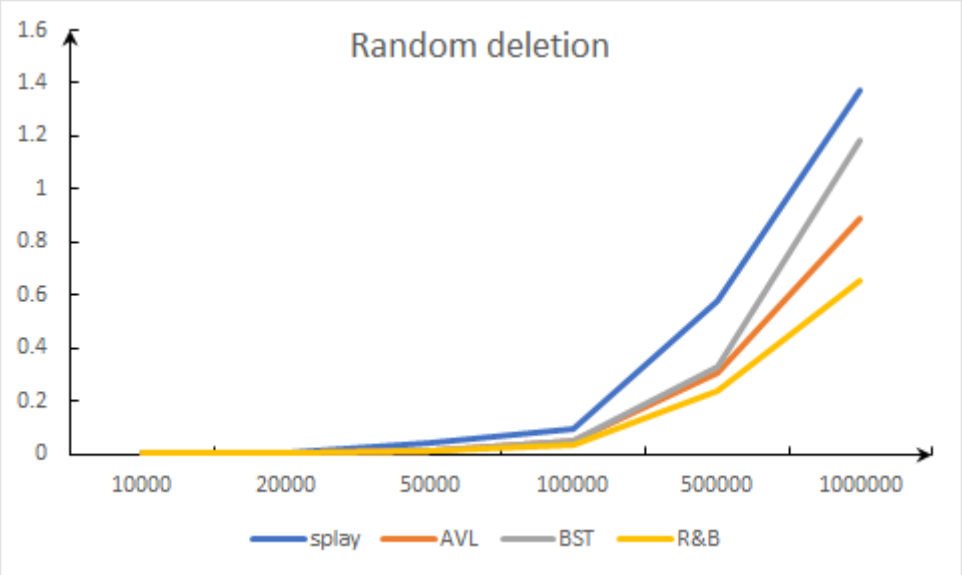
The splay tree has the largest time cost and the highest height, for it has no strict balance requirements and need to do much extra splay operation. BST don't adjust the tree, so it's height may also be big, which makes the operation difficult. AVL tree and R&B tree has almost the same time cost.

Random query:



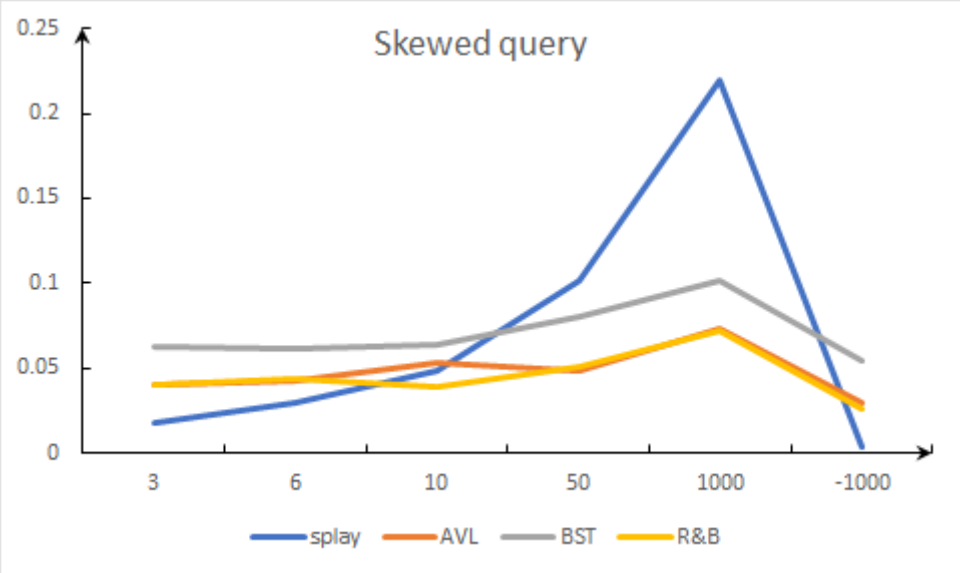
AVL tree, BST, R&B tree implement the function all in iterative way. So their time cost are almost the same and small. But splay tree need to do splay operation when query, and the splay is implemented in recursive way, so the time cost is much bigger.

Random deletion:



As query operation, splay tree need to do splay operation each deletion. BST has a bigger height. AVL tree and R&B tree are well balanced but AVL tree has a more strict standard so it will cost more time.

Skewed query:



We generate 1000000 random numbers whose values are among N numbers.

We can see that, as N increases, the splay tree loses its advantage. The -1000 means we order the generated numbers (among 1000 numbers) , the time cost of splay tree is greatly reduced. Then we get that splay trees are only good for finding the same data or a very small amount of data in a short time. But the time costs of other trees are also reduced. We guess that because our PC's own ability. PC may use the same principle as splay tree when run our program.

5. Conclusion

In this project, we have deepened our understanding of all kinds of trees. As for result, R&B tree has the best performance, splay tree do well in skewed query but not in other operations, the performance of BST is greatly influenced by the data type.