# Comparing Different Binary Search Trees

**Group：3**

**Author: Yi Jiang, Yuzhuo Yang, Juntao Huang**

**2023-03-26**

# 0.introduction

In AVL trees, the balance criterion is on the height of subtrees. Some other trees may achieve balance by restricting the weight of subtrees. One of such trees, namely the BB[α] trees (BB stands for bounded balance), is introduced in problem 17-3 of *Introduction to Algorithms (third edition)*.

In this project, we are required to do **the followings**.

(1)Give a theoretical analysis for BB[α] trees. That is, we should solve all the questions in problem 17-3.

(2)Implement the basic BST, the AVL tree, the splay tree, and the BB[α] tree.

(3)Run experiments to compare their performance.

**Experiments**

We should insert N elements into an empty tree in increasing order or in random order, and measure how the following three quantities change against N. When we insert in random order, we should use the expectation of these quantities.

- total time cost by the insertions
- total number of times that the tree (AVL and BB[α] only) becomes imbalanced during the insertions
- average depth of nodes after insertions

We may also try a random mixed sequence of insertions and deletions, and measure (against the length of sequence of operations) the expected total number of times that the tree (AVL and BB[α] only) becomes imbalanced.

# 1. BB[α] trees

problem 17-3 of *Introduction to Algorithms (third edition)*

Consider an ordinary binary search tree augmented by adding to each node x the attribute x.size giving the number of keys stored in the subtree rooted at x. Let α be a constant in the range [0.5,1). We say that a given node x is α-balanced if x.left.size no greater than α*x.size and x.right.size no greater than α*x.size. The tree as a whole is α-balanced if every node in the tree is α-balanced. The following amortized approach to maintaining weight-balanced trees was suggested by G. Varghese.

**a.** A 1/2-balanced tree is, in a sense, as balanced as it can be. Given a node x in an arbitrary binary search tree, show how to rebuild the subtree rooted at x so that it becomes 1/2-balanced. Your algorithm should run in time Θ(x.size),and it can use O(x.size) auxiliary storage.

**Answer:**

Because we can use O(x.size) extra space and ,O(x.size) time, and we know that for a BST, its in-order traversal is an ascending-order array, we can rebuild a 1/2-balanced BST according to this array(recursively select the middle element as the root and do the same to the left-half and right-half), we used O(x.size) extra space and O(x.size) time. (O(x.size) for traversal, O(x.size) for rebuilding tree).

**b.** Show that performing a search in an n-node α-balanced binary search tree takes O(lg n) worst-case time.

**Answer:**

We can prove it by proving the height of the tree is *O(log N)*.Because in a α-balanced tree, every node obeys that

$$x.left.size < \alpha * x.size$$

And

$$x.right.size < \alpha * x.size.$$

Suppose that the height of the tree is h.

$$root.left.size < \alpha * root.size$$

$$root.right.size > root.size - \alpha * root.size - 1$$

so

$$root.right.size > (1 - \alpha)root.size - 1$$

suppose that

$$root.size < b^h$$

$$root.right.size < \alpha * root.size < \alpha * b^h$$

but

$$root.right.size > (1 - \alpha)root.size - 1,$$

which is a contradictory. So for a α-balanced tree of height h, there must be more than $b^h$ nodes. So performing a search in an n-node α-balanced binary search tree takes O(lg N) worst-case time.

For the remainder of this problem, assume that the constant α is strictly greater than 1/2. Suppose that we implement INSERT and DELETE as usual for an n-node binary search tree, except that after every such operation, if any node in the tree is no longer α-balanced, then we "rebuild" the subtree rooted at the highest such node in the tree so that it becomes 1/2-balanced.We shall analyze this rebuilding scheme using the potential method. For a node x in a binary search tree T , we define

$$\Delta(\mathrm{x}) = |x.left.size - x.right.size|$$

and we define the potential of T as

$$\Phi(T) = c\Sigma\Delta(x)$$

where c is a sufficiently large constant that depends on α.

**c.** Argue that any binary search tree has nonnegative potential and that a 1/2-balanced tree has potential 0.

**Answer:**

Since $\Delta(\mathrm{x}) = |x.left.size - x.right.size| > 0$.Potential function which is the sum of them is still nonnegative.

Since for a 1/2 balanced BST, left.size is equal to right.size, $\Delta(\mathrm{x}) = |x.left.size - x.right.size| = 0$, so it has potential 0.

**d.** Suppose that m units of potential can pay for rebuilding an m-node subtree. How large must c be in terms of α in order for it to take O(1) amortized time to rebuild a subtree that is not α-balanced?

**Answer:**

$$O(1) = m + \Phi(Di) - \Phi(Di - 1)$$

We take O(1) as 0 because m is O(n), and O(1) is too small when facing O(n). then we get:

$$m + \Phi(Di) = \Phi(Di - 1)$$

Since the potential function is not negative, we have:

$$m < \Phi(Di - 1)$$

And we have

$$\Delta(\mathrm{x}) = |x.left.size - x.right.size| \geq \alpha m - \big((1-\alpha)m - 1\big)$$

So

$$\Delta(\mathrm{x}) \geq (2\alpha - 1)m + 1 > m \times \frac{1}{c}$$

And then

$$(2\alpha - 1) + 1/m > \frac{1}{c}$$

Since m is no small than 1,we have

$$2\alpha > \frac{1}{c}$$

That is

$$c > \frac{1}{2\alpha}$$

**e.** Show that inserting a node into or deleting a node from an n-node α-balanced tree costs O(lg N) amortized time.

**Answer:**

Regardless of insertion or deletion, the find costs O(lg N) time, so we only need to deal with the balancing part. from the analysis in d, re-balance the tree costs O(1)(average).By inserting or deleting, we will make at most O(lg N) nodes unbalanced, we only need to rebalance O(lg N) nodes. So the time complexity is O(lg N).

# 2. theoretical comparison of BST

### BST

The worst-case complexity of a single search in a BST is clearly $O(n)$, considering the case of a completely skewed tree (which can be easily constructed using an input sequence of increasing/decreasing order). In this case the BST will degenerate into a linear list.

And the worst-case complexity of a single insertion/deletion in BST is obviously still $O(n)$, since they consists of a `search()` operation and other operations such as `create_node()` and `swap()`, which only takes $O(1)$ time. So the result should be $O(n) + O(1) = O(n)$.

However, this worst-case scenario is very unlikely to happen, especially if the tree is balanced. On average, the height of a balanced binary search tree grows logarithmically with the number of nodes, so the average time complexity of an insert operation is $O(\log n)$. This means that if we perform a large number of insert operations on a binary search tree, the average time complexity of each insert operation would be $O(\log n)$. Hence the average time complexity for single insertion/deletion is also $O(\log n)$.

Notice that above is NOT an amortized analysis on this data structure, for we're not sure about the exact content of the operation sequences, thus unable to construct a valid potential function. The conclusion is only based on a vague estimation on the height of a BST with an input set of size large enough.

**AVL tree**

The worst-case time complexity of a single search in an AVL tree is $O(\log n)$ since in this data structure, the maximum height of tree $h_{max} = O(\log n)$ (proven in class using mathematical induction). While the tree still has all BST properties, going down from the root to target node, the total time taken by searching for an element won't go any further than $O(\log n)$. This conclusion also holds for a single insertion in AVL tree, since after finding the position to be inserted into and linking the newly created node with the tree, it only takes limited time of rotation(s) (1 in LL/RR cases and 2 in LR/RL cases) to restore balance. The tree is expected to immediately restore balance after 1 or 2 rotations (also proven in class). Hence the time to perform a single rotation is $O(1)$, the total time taken should be $O(log\ n) + c \cdot O(1) = O(log\ n), c = 1,2$.

But the process gets a little different in deletions. The first few steps is still the same as in BST until the balance-restoration. The node deletion can possibly result in the imbalance of all its ancestors, each of which we'll have to perform a single/double rotation on, and that is a total of $h = O(\log n)$. Hence the overall time complexity is $O(\log n) + O(\log n) = O(\log n)$.

**Splay tree**

The worst-case time complexity of a single search in an Splay tree is $O(n)$ since in this data structure, the maximum height of tree $h_{max} = O(n)$ (inserting n nodes in increasing order and find the largest key). While the tree still has all BST properties, going down from the root to target node and rotate the node which was recently found to root, the amortized time taken by searching for an element won't go any further than $O(\log n)$ as proved in class.

This conclusion also holds for a single insertion in Splay tree, since in this data structure, the maximum height of tree $h_{max} = O(n)$ (inserting n nodes in increasing order and then insert the largest key). While the tree still has all BST properties, going down from the root to target node and rotate the node which was recently found to root, the amortized time taken by searching for an element won't go any further than $O(\log n)$ as proved in class.

But the process gets a little complex in deletions. The deletion is first Find X, then Remove X, then FindMax ($T_L$) and finally Make $T_R$ the right child of the root of $T_L$. The worst case, time complexity is $O(n) + O(1) + O(n) + O(1) = O(n)$

since in this data structure, the maximum height of tree $h_{max} = O(n)$ as shown above. But in amortized cases, the tree time complexity is $O(log\ n) + O(1) + O(log\ n) + O(1) = O(log\ n)$ since the amortized time of splaying a node is $O(\log n)$.

**Comparison(>means time complexity from large to small)**

**Insert**

Worst case:

BST ≈ Splay tree > AVL tree

Amortized case:

BST ≈ BB[α] tree ≈ Splay tree ≈ AVL tree

**Delete**

Worst case:

BST ≈ BB[α] tree ≈ Splay tree > AVL tree

Amortized case:

BST ≈ BB[α] tree ≈ Splay tree ≈ AVL tree

**Search/find**

Worst case:

BST ≈ Splay tree > AVL tree ≈ BB[α] tree

Amortized case:

BST ≈ BB[α] tree ≈ Splay tree ≈ AVL tree

# 3. Description of experiments

### how experiments are designed

BST/AVL tree/Splay tree/BB[α] tree

The testing program contains 3 basic operations of tree: find_Key, insert and delete. Each of them are strictly constructed according to the process described in the textbook *data structure and algorithm analysis in C*. As for BB[α] tree, the detailed information is in appendix.

random order only insertion: Computer will generate a specified number of random number and store them in a array, and insert them one by one into an empty tree.

random order insertion + random order deletion: Generate random number the same as the last one, but generate random array index later and delete the node in the tree according to the generated index.

increasing order only insertion: Generate array {1,2,3,4,5,6,......}, and insert the element one by one in an empty tree.

increasing order insertion + same order deletion, same insertion as the last one, delete one by one in a sequential order.

**how testing data are generated**

The random input data is generated via the rand() function in C standard library with srand(time(NULL)), stored in an array, and printed along with another outputs at the end of the program. Operation sequence is the same with the generated random array, first all insertions, then all deletions.

**how results are obtained**

The time-measurement is done by the time() function, stored in a variable with a type "time_t". The timer will start before the first **operation** and terminates immediately after the last **operation**, ignoring the time wasted to generating insertion data/deletion data. If time interval is short, we let the program run for a certain number of times **repeatedly** to get the total time and then divide the number of runs to get one pass of the run time. Average depth will be calculated by **level order traversal** after the timer is done.

# 4. Results of experiment and analysis of these results

**Result**

**BST insert in random order**

| Node number | total time cost by insertions(ms) | average depth of nodes after insertions |
|---|---|---|
| 100 | 0.923 | 7.560 |
| 1000 | 5.711 | 27.896 |
| 10000 | 436.860 | 173.589 |

**Splay tree insert in random order**

| Node number | total time cost by insertions(ms) | average depth of nodes after insertions |
|---|---|---|
| 100 | 0.035 | 6.540000 |
| 1000 | 0.924 | 11.435000 |
| 5000 | 4.000 | 14.833800 |
| 10000 | 7.670 | 16.492600 |
| 25000 | 19.567 | 18.619360 |
| 50000 | 31.533 | 18.753652 |
| 75000 | 60.076 | 19.735587 |

| | | |
|---|---|---|
| 100000 | 79.000 | 21.762389 |

**Splay tree insert in random order and delete in random order**

| Node number | total time cost by insertions(ms) | average depth of nodes after insertions |
|---|---|---|
| 100 | 0.333 | 5.630000 |
| 1000 | 1.924 | 10.678000 |
| 5000 | 7.592 | 14.010600 |
| 10000 | 20.324 | 15.970800 |
| 25000 | 44.290 | 16.871365 |
| 50000 | 80.533 | 17.993652 |
| 75000 | 134.430 | 19.735587 |
| 100000 | 198.953 | 24.870155 |

**Splay tree insert in increasing order**

| Node number | total time cost by insertions(ms) | average depth of nodes after insertions |
|---|---|---|
| 100 | 0.015 | 49.500000 |
| 1000 | 0.170 | 499.50000 |
| 5000 | 0.440 | 2449.5000 |
| 10000 | 1.040 | 4999.5000 |
| 25000 | 2.284 | 12499.500 |
| 50000 | 5.365 | 24999.500 |
| 75000 | 7.199 | 37499.500 |
| 100000 | 10.137 | 49999.500 |

**AVL tree insert in random order and delete in random order**

| Node number | total time cost by insertions(ms) | average depth of nodes after insertions | imbalanced times |
|---|---|---|---|
| 100 | 0.018 | 4.25306 | 100 |
| 1000 | 0.133 | 7.6645 | 972 |
| 5000 | 0.694 | 9.10708 | 4455 |
| 10000 | 1.407 | 9.65366 | 8157 |
| 25000 | 2.085 | 10.01694 | 11306 |
| 50000 | 3.169 | 10.1056 | 16989 |
| 75000 | 6.139 | 10.51023 | 29927 |
| 150000 | 9.614 | 10.55499 | 42506 |

**BB[α] tree insert in random order**
**balance factor α = 0.75, delete factor = 0.5**

| Node | total time cost by | average depth of nodes | imbalanced times |
|---|---|---|---|

| number | insertions(second) | after insertions | |
|---|---|---|---|
| 100 | 0.000030 | 6.810000 | 5 |
| 1000 | 0.004000 | 11.435000 | 31 |
| 5000 | 0.101000 | 14.833800 | 98 |
| 10000 | 0.541000 | 16.492600 | 211 |
| 25000 | 3.770000 | 18.619360 | 319 |
| 50000 | 13.96800 | 18.753652 | 614 |
| 75000 | 26.64900 | 19.735587 | 865 |
| 100000 | 40.44700 | 21.762389 | 1003 |

**BB[α] tree insert in random order and delete in random order**
**balance factor α = 0.75, delete factor = 0.5**

| Node number | total time cost by insertions(second) | average depth of nodes after insertions | imbalanced times |
|---|---|---|---|
| 100 | 0.000004 | 6.620000 | 6 |
| 1000 | 0.009000 | 10.115000 | 41 |
| 5000 | 0.210000 | 12.402800 | 109 |
| 10000 | 1.089000 | 13.332600 | 235 |
| 25000 | 8.021000 | 14.612000 | 377 |
| 50000 | 28.122000 | 15.567000 | 685 |
| 75000 | 54.121000 | 16.197000 | 897 |
| 100000 | 80.912000 | 16.611000 | 1126 |

**BB[α] tree insert in increasing order**
**balance factor α = 0.75, delete factor = 0.5**

| Node number | total time cost by insertions(second) | average depth of nodes after insertions | imbalanced times |
|---|---|---|---|
| 100 | 0.001000 | 6.64 | 7 |
| 1000 | 0.013000 | 9.96 | 31 |
| 5000 | 0.198000 | 12.29 | 93 |
| 10000 | 0.825000 | 13.29 | 212 |
| 25000 | 4.581000 | 14.60 | 343 |
| 50000 | 17.843000 | 15.60 | 586 |
| 75000 | 53.511000 | 16.20 | 802 |
| 100000 | 86.842000 | 16.60 | 1087 |

**BB[α] tree insert in increasing order and delete in same order**
**balance factor α = 0.75, delete factor = 0.5**

| Node number | total time cost by insertions(second) | average depth of nodes after insertions | imbalanced times |
|---|---|---|---|

| | | | |
|---|---|---|---|
| 100 | 0.001000 | 6.64 | 9 |
| 1000 | 0.026000 | 9.96 | 39 |
| 5000 | 0.606000 | 12.29 | 99 |
| 10000 | 2.105000 | 13.29 | 224 |
| 25000 | 11.225400 | 14.60 | 389 |
| 50000 | 37.852000 | 15.60 | 611 |
| 75000 | 109.562000 | 16.20 | 851 |
| 100000 | 189.196000 | 16.60 | 1194 |

**Analysis**

- When data is almost in order, the Splay tree is faster than BST, AVL Tree, BB[α] tree when inserting/deleting since it can splay the node recently inserted to the root and make the next insertion more easier.
- When data is almost random, the BB[α] tree is faster than BST, Splay tree when inserting/deleting since it can rebuild part of itself to CBST and maintain the height of the tree stable(O(log N)) and moreover, it use lazy deletion to simply the operation.
- AVL Tree is faster than Splay tree when inserting data is almost random since the find operation don't save time much in random order insertions.
- For the average depth of nodes, the balanced BST is more stable than imbalanced BST since they can balance themselves when inserting.
- For the imbalanced times, the BB[α] tree is more stable than AVL Tree since the BB[α] tree can rebuild part of itself to CBST when it is imbalanced to get itself more balanced.

# 5. Conclusion

- When data is almost in order, the Splay tree is faster than BST, AVL Tree, BB[α] tree when inserting/deleting.
- When data is almost random, the BB[α] tree is faster than BST, AVL Tree, Splay tree when inserting/deleting.
- For the average depth of nodes, the balanced BST is more stable than imbalanced BST.
- For the imbalanced times, the BB[α] tree is more stable than AVL Tree.

# 6. Appendix

**BB[α] Tree**

```
AlphaBalancedTreePtr CreateNode(int value)
{
    AlphaBalancedTreePtr node
= (AlphaBalancedTreePtr)malloc(sizeof(struct AlphaBalancedTree));
    node->isDeleted = false;
    node->value = value;
    node->childNum = 0;
    node->deletedChildNum = 0;
    node->left = NULL;
    node->right = NULL;
    node->parent = NULL;
    return node;
}
AlphaBalancedTreePtr Find(AlphaBalancedTreePtr root, int value)
{
    if (root == NULL || root->isDeleted)
        return NULL;
    if (root->value == value && !root->isDeleted)
        return root;
    if (value < root->value)
        return Find(root->left, value);
    else
        return Find(root->right, value);
}
AlphaBalancedTreePtr  FindIncludeDeleted(AlphaBalancedTreePtr  root,
int value)
{
    if (root == NULL)
        return NULL;
    if (root->value == value && !root->isDeleted)
        return root;
    if (value < root->value)
        return Find(root->left, value);
    else
        return Find(root->right, value);
}
AlphaBalancedTreePtr Delete(AlphaBalancedTreePtr root, int value)
{
    root = DeleteWithoutAdjust(root, value);
    root = Adjust(root);
    return root;
```

```
}
AlphaBalancedTreePtr  DeleteWithoutAdjust(AlphaBalancedTreePtr  root,
int value)
{
    AlphaBalancedTreePtr temp = FindIncludeDeleted(root, value);
    if (temp == NULL)return root;
    bool isDeleted = temp->isDeleted;
    if (root == NULL)return root;
    else if (value < root->value)
    {
        root->left = DeleteWithoutAdjust(root->left, value);
        isDeleted ? root->deletedChildNum++ : root->childNum--;
    }
    else if (value > root->value)
    {
        root->right = DeleteWithoutAdjust(root->right, value);
        isDeleted ? root->deletedChildNum++ : root->childNum--;
    }
    else
        root->isDeleted = true;
    return root;
}
AlphaBalancedTreePtr Insert(AlphaBalancedTreePtr root, int value)
{
    root = InsertWithoutAdjust(root, value, NULL);
    root = Adjust(root);
    return root;
}
AlphaBalancedTreePtr  InsertWithoutAdjust(AlphaBalancedTreePtr  root,
int value, AlphaBalancedTreePtr parent)
{
    AlphaBalancedTreePtr temp = FindIncludeDeleted(root, value);
    if (temp != NULL && !temp->isDeleted)return root;
    bool isDeleted = (temp != NULL);
    if (root == NULL)
    {
        root = CreateNode(value);
        root->parent = parent;
    }
    else if (value < root->value)
    {
        root->left = InsertWithoutAdjust(root->left, value, root);
        isDeleted ? root->deletedChildNum-- : root->childNum++;
    }
```

```
    else if (value > root->value)
    {
        root->right = InsertWithoutAdjust(root->right, value, root);
        isDeleted ? root->deletedChildNum-- : root->childNum++;
    }
    else
        root->isDeleted = false;
    return root;
}
AlphaBalancedTreePtr Adjust(AlphaBalancedTreePtr root)
{
    if (root == NULL)return root;
    else if (isBalanced(root))
    {
        if (root->left != NULL)
            root->left = Adjust(root->left);
        if (root->right != NULL)
            root->right = Adjust(root->right);
        return root;
    }
    else
    {
        AlphaBalancedTreePtr temp = root->parent;
        root = Rebuild(root);
        root->parent = temp;
        if (root->left != NULL)root->left = Adjust(root->left);
        if (root->right != NULL)root->right = Adjust(root->right);
        return root;
    }
}
AlphaBalancedTreePtr Rebuild(AlphaBalancedTreePtr root)
{
    if (root->childNum == 1)return root;
    int* inorderTraversal = GetInorderTraversal(root);
    root = BuildTotallyBalancedTree(inorderTraversal, root->childNum -
root->deletedChildNum + 1);
    UpdateUndeletedChildNum(root);
    return root;
}
AlphaBalancedTreePtr  BuildTotallyBalancedTree(int*  inorderTraversal,
int num)
{
    AlphaBalancedTreePtr node = CreateNode(inorderTraversal[(num - 1)
/ 2]);
```

```
    if (num == 0)     return NULL;
    node->left = BuildTotallyBalancedTree(inorderTraversal, (num - 1) / 2);
    node->right = BuildTotallyBalancedTree(inorderTraversal + (num - 1) / 2 + 1, num - (num - 1) / 2 - 1);
    if (node->left != NULL)node->left->parent = node;
    if (node->right != NULL)node->right->parent = node;
    return node;
}
void InorderTraversal(AlphaBalancedTreePtr root, int* result, int* currentIndex)
{
    if (root == NULL || root->isDeleted)return;
    if (root->left != NULL)
        InorderTraversal(root->left, result, currentIndex);
    result[*currentIndex] = root->value;
    (*currentIndex)++;
    if (root->right != NULL)
        InorderTraversal(root->right, result, currentIndex);
    return;
}
float GetBalanceFactor(AlphaBalancedTreePtr root)
{
    if (root == NULL)return 0;
    return root->left == NULL ? 0 : (float)(root->left->childNum + 1) / (root->childNum + 1);
}
float GetDeletedScale(AlphaBalancedTreePtr root)
{
    if (root == NULL)return 0;
    return (float)root->deletedChildNum /
 (root->childNum + root->deletedChildNum + 1);
}
int* GetInorderTraversal(AlphaBalancedTreePtr root)
{
    int* inorderTraversal = malloc((root->childNum - root->deletedChildNum + 1) * sizeof(int));
    int* index = malloc(sizeof(int));*index = 0;
    InorderTraversal(root, inorderTraversal, index);
    return inorderTraversal;
}
int* GetRandomIntArrayWithRange(int num, int min, int max)
{
    if (num == 0)
```

```
    if (num == 0)return NULL;
    srand((unsigned)time(NULL));
    int* result = malloc(num * sizeof(int));
    for (int i = 0; i < num; i++)
        result[i] = rand() % (max - min + 1) + min;
    return result;
}
int UpdateUndeletedChildNum(AlphaBalancedTreePtr root)
{
    int childNum = 0;
    if(root == NULL)return 0;
    childNum += UpdateUndeletedChildNum(root->left);
    childNum += UpdateUndeletedChildNum(root->right);
    root->childNum = childNum;
    return childNum+1;
}
bool isBalanced(AlphaBalancedTreePtr node)
{
    if (node == NULL || node->childNum == 0 || node->childNum == 1)
        return true;
    float balanceFactor = GetBalanceFactor(node);
    return balanceFactor > targetBalanceFactor && balanceFactor < 1 -
targetBalanceFactor;
}
```