Part of Lab1 report

March 2023

1 AVLTree

1.1 Implementation

Before we start analyse the performance of the AVLTree, we need to talk about the implementation of the AVLTree.

And in the operations of AVLTree, 'Rotation' is the most important operation. Here are several examples of the Rotation:

1.1.1 Rotation

Because the rotation can be easily implemented by change the pointers, it has a time cost of O(1).

1.1.2 findKey

According to the characteristic of the AVL binary search tree, the worst search time will be O(h), where h is the height of the AVLTree. Thus time cost of findkey will be O(logn).

1.1.3 insertion

Firstly, we need to find the position to insert the new element, which costs O(logn). There are 4 imbalanced conditions which need to be re-balanced, that is, LL, LR, RR and RL. After rotation, BF will decreased by 1, and all its ancestors will be influenced. Thus the maximum rotation number will be 2. That's still O(1).

Finally, the total time cost of insertion is $O(logn)$.

1.1.4 deletion

Firstly, we need to find the position of the key to delete, which costs $O(logn)$. But the deletion of the key may cause the imbalance of the ancestors, and the worst case is that after balancing the subtree, the ancestor was still imbalanced. Thus, The balance operation will be at most **logn** times.

Finally, the total time cost will be $O(logn) + O(1) \times O(logn) = O(logn)$

1.2 Performance

1.2.1 Sequence

After every 2 insertion, the tree will be imbalanced and need to re-balance. The total time cost of insertion will be

$$T(findKey) + T(balance) = O(1 + 2 + 2 + 3 + 3 + 4 + ... + (logn) - 1 + logn)$$

$$+O(1) \times O(\frac{n}{3}) = O(log^2n) + O(n) = O(n)$$

1.2.2 Amortized

$$O(logn)$$

total time cost:

$$O(mlogn)$$

where m is the number of operations

2 SplayTree

2.1 Implementation

Same as the AVLTree, 'Rotation' is the most important operation in the SplayTree as well.

2.1.1 Rotation

Same as the AVLTree.

2.1.2 findKey

Same as the AVLTree.

2.1.3 Splay

To rotate the new element or newly found key to the root, we have a time cost of $O(1) \times O(h)$, Then because $h \leq logn$, the time cost of Splay will be log(n).

2.1.4 insertion

Firstly, find the position to insert the new element, which costs $O(logn)$ times. After insertion, the element need to be splayed to the root, so the total time cost is $O(logn)$

2.1.5 deletion

Firstly, findKey($O(logn)$) and Splay($O(logn)$) it to the root, then delete($O(1)$) it.

The total time cost of deletion is $O(logn)$.

2.2 Performance

2.2.1 Sequence

Every time after insertion, the new element will be the right son of the root, **findKey** only costs O(1), and then **Splay** it to the root also costs only $O(1)$, thus the average cost of insertion is $O(1)$, thus the total cost is $O(n)$.

2.2.2 Amortized

$$O(logn)$$

total time cost:

$$O(mlogn)$$

Potential Function:

$$\Phi(u) = rank(u) = log(size(u))$$

$$\Phi(Tree) = \sum_{u \in T} \Phi(u)$$

1) for single rotate: $u \to v$

$$\hat{c_i} = c_i + \Delta\Phi(u) + \Delta\Phi(v) < 1 + \Delta\Phi(v)$$

2) for zig-zag/zag-zig: $u \to v \to w$

$$\hat{c_i} = c_i + \Delta\Phi(u) + \Delta\Phi(v) + \Delta\Phi(w)$$

$$= c_i + r(u') - r(u) + r(v') - r(v) + r(w') - r(w)$$

$$= c_i + r(u') + r(v') - r(v) - r(w)$$

$$< c_i + r(u') + r(v') - 2r(w)$$

$$< c_i + 2log(\frac{size(u') + size(v')}{2}) - 2r(w)$$

$$= c_i + 2(r(w') - 1) - 2r(w)$$

$$= 2(r(w') - r(w))$$

$$= 2\Delta\Phi()$$

3) for zig-zig/zag-zag: $u \to v \to w$

$$\hat{c_i} = c_i + \Delta\Phi(u) + \Delta\Phi(v) + \Delta\Phi(w)$$

$$
\begin{aligned}
&= c_i + r(u') - r(u) + r(v') - r(v) + r(w') - r(w) \\
&= c_i + r(u') + r(v') - r(v) - r(w) \\
&< c_i + r(u') + r(v') - 2r(w) \\
&< c_i + r(u') + r(w') + r(w) - 3r(w) \\
&< 3(r(w') - r(w)) \\
&= 3\Delta\Phi(w)
\end{aligned}
$$

the amortized time cost of a single operation is O(logn)

3 Testing

3.1 Data generator

3.1.1 sequence

Sequentially insert number 1 to n, calculate the time spent.

3.1.2 random

Use func rand() to rearrange the sequence of 1 to n to generate a non-sequential array of input. Mark some of them insertions and the others deletion, calculate the time spent.

3.2 Result

Use the chrono lib to calculate the time spend during the insertion/deletion. Print the total time of different trees and compare the imbalanced times in a file.

3.2.1 Graph

From the output of different scales, we can draw graph for different conditions:
1) Sequential insertions: **insert pic1**
We can see:
1.BST $O(n^2)$
2.AVL $O(n)$
3.Splay $O(n)$
4. BBalpha $O(n)$
2) Random mixed operations: **insert pic2**
We can see:
1.BST $O(mlogn)$
2.AVL $O(mlogn)$
3.Splay $O(mlogn)$
4.BBalpha $O(mlogn)$
3) different α for $BB[\alpha]$ Tree:

3.2.2 Analysis

The data in the graph shows the correctness of the theoretical performance of the trees, and we can find the $BB[\alpha]$ Tree has an excellent performance comparing to the splay tree and the avl Tree and have easier operations if choose a good α.

4 Conclusion

During the experiment, we find that $BB[\alpha]$ Tree have a efficiency close to the AVLTree, but have easier operations than other trees.