Performance comparison of binary search trees

2021320117 Bae Minseong

I. Theoretical Background

1. Binary Search Tree (BST)

Binary search tree is a binary tree such that every element has a non-duplicate key and keys in the left/right subtree are smaller/larger than the key of the root. Also, left and right subtrees are binary search tree.

Operations in binary search tree like searching, inserting, and deleting follow the time complexity of O(h), when h is the height of binary search tree. The best case of the height of a binary search tree is O(log n) such as complete/full binary tree. In contrast, the worst case of the height of a binary search tree is O(n) such as a skewed tree.

2. AVL Tree

AVL tree is a kind of dynamic balanced binary search tree. It keeps its height O(log n) for searching, inserting, and deleting. Using a balance factor, a difference between the height of left and right subtree, the tree automatically restructure itself by the rotation. The height of an AVL tree is O(log n), so major operations can be done in O(log n) time.

II. Simple Code Explanation for main.cpp

For the test of performance of BST and AVL tree, I implemented a test function.

First, I defined the **struct time_set**, the return value of test function. It contains four elements; each float type variable represents the time for operations in the binary search tree with skewed and random dataset. (random insertion/find + skewed insertion/find)

```
struct time_set{

    float r_insert;
    float r_find;
    float s_insert;
    float s_find;

};
```

In the test function, using **std::rand()**, I create a random dataset. It contains **nElem (number of elements)** nonduplicate elements (Each number should be once appeared) in the range of [1, nElem]. To make the dataset nonduplicate, I used a Boolean array to check numbers already appeared. Array **find_key** is the dataset used for finding operation test. I didn't fix the range of find_key so if the key is bigger than nElem,

finding operation will be resulted as unsuccessful search. Additionally, because the range of the dataset is fixed as [1, nElem], I made a skewed dataset with a just simple for loop. So, the skewed dataset will be made as the form of [1, 2, 3, … , nElem – 1, nElem].

```
int *key = new int[nElem];
int *find_key = new int[nElem];
float *val = new float[nElem];
bool *isInserted = new bool[nElem+1]; //check duplicate insertion

for(int i=0;i<nElem+1;i++){
    isInserted[i] = false;
}

std::srand(std::time(0));

for(int i=0;i<nElem;++i){
    find_key[i] = std::rand();
    while(true){
        int tmp = (std::rand() % nElem) + 1; //1 ~ nElem
        if(!isInserted[tmp]){
            isInserted[tmp] = true;
            key[i] = tmp;
            val[i] = (float)std::rand() / RAND_MAX * 20000;
            break;
        }
    }
}
```

```
//make skewed dataset

for(int i=0;i<nElem;i++){
    key[i] = i+1;
}
```

I defined two binary search trees for the skewed and random dataset, **st_skewed** and **st_random**. Using **clock()**, I calculated the time of insertion and finding for each four cases, insertion and finding for skewed and random dataset. After the calculation, I saved four calculated time to time_set variable **res** and return it**.**

```
clock_t tm;
tm = clock();
for(int i=0; i<nElem; i++){
    st_random.insert(key[i], val[i]);
}
tm = clock() - tm;
res.r_insert = ((float)tm / (float)CLOCKS_PER_SEC);
```

In the main function, I called test() for BST and AVL tree implemented in Searchtree.h and AVLTree.h. I used the template in the definition of test() so I can just simply call the function like this.

```
time_set test_res_BST = test<SearchTree<EntryType> >(test_size_BST[s]);
```

I chose the size of testcase like this. In the test Linux server, operations In BST for skewed dataset of size bigger than 100000 didn't work because of memory overflow, so I excluded the skewed case of size 1000000 using **std::is_same<>**.

```
if(nElem < 1000000 || std::is_same<TreeType, AVLTree<EntryType> >::value)
```

```
int test_size[5] = {1000, 10000, 50000, 100000, 1000000};

cout << "< BST Test >" << endl;
for(int s=0;s<5;s++){

    time_set test_res_BST = test<SearchTree<EntryType> >(test_size[s]);

    cout << "Test Size : " << test_size[s] << endl;
    cout << "BST Insert (Random) takes " << test_res_BST.r_insert << " seconds." << endl;
    cout << "BST Find (Random) takes " << test_res_BST.r_find << " seconds." << endl;
    if(s<4){
        cout << "BST Insert (Skewed) takes " << test_res_BST.s_insert << " seconds." << endl;
        cout << "BST Find (Skewed) takes " << test_res_BST.s_find << " seconds." << endl;
    }

}
```

III. Results

This is the table of running time for each case. (Codes are run in the provided test Linux server and the unit of each figure is a second.)

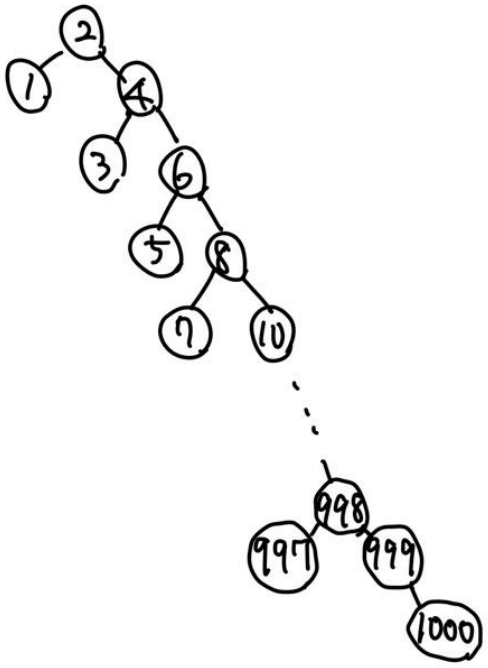
Input size		1000	10000	50000	100000	1000000
Random	BST-Insert	0.001126	0.003964	0.023715	0.053344	1.13306
	BST-Find	0.000119	0.001995	0.011879	0.020292	0.195821
	AVL-Insert	0.001292	0.017722	0.103324	0.228732	3.2654
	AVL-Find	0.000254	0.002412	0.015272	0.034387	0.378261
Skewed	BST-Insert	0.006901	0.694685	18.0846	86.8567	-
	BST-Find	0.01346	1.38324	37.4861	189.557	-
	AVL-Insert	0.001414	0.018335	0.10492	0.219548	2.59639
	AVL-Find	0.000184	0.002585	0.016503	0.034469	0.384037

IV. Analysis

Before analyzing the actual figure, let's think about the expected structure of each binary search tree.

For the random dataset of size n, the expected height of BST will be O(log n) because in the each level of BST the probability that the newly inserted entry is included in left and right subtree is both 1/2, so its expected height will be O(log n). But for the skewed dataset, the structure of BST will be right-skewed n-height tree because the order of insertion is [1, 2, …, n] so the height will be O(n).

For both skewed and random dataset of size n, the height of AVL tree will be maintained as O(log n) because of its height-balancing property. Especially for skewed dataset of size n, the AVL tree will be structured like this picture.



For the random dataset, we can find out insertion for BST approximately follows O(log n). Insertion for AVL tree also approximately follows O(log n) except size 1000000 but I think this error is caused by the order of number in random dataset size of 1000000, maybe it required lots of rotation. And we can also find out insertion time of AVL tree is bigger than insertion time of BST for every input size because of additional rotation in restructuring AVL tree.

Unfortunately, searching time for both tree in random dataset looks like O(n), but this phenomenon would be caused by the procedure for choosing find key. We chose the find key by random number generator, so the range of generated number is much bigger than the range [1, nElem]. It means most of the search in this experiment would be unsuccessful search. If we choose the find key in [1, nElem], the searching time will be similar to insertion time, which is approximately O(log n). But we can find out the difference between the searching time for AVL tree and BST is very small comparing to the difference in insertion time, that means AVL tree and BST would have similar height.

In the skewed dataset, we can't easily find out exact time complexity of insertion and searching time. However, we can find out that the insertion/searching time in BST is much bigger than the time in random dataset while the insertion/searching time in AVL tree is similar to the time in random dataset. This is due to the restructuring procedure in AVL tree. Especially, when the input size is bigger than 50000, searching time in BST is tremendously big, because the height of skewed BST is O(n) and most of the searching case in big input size dataset is unsuccessful search.

All these things considered, we can conclude that **the performance of AVL tree is**

better than the performance of (normal) binary search tree, especially for big and skewed data.