CS 310

Assignment 322

March 22, 2022

Jingbo Wang

The `avl.h` algorithm given in the assignment could build an "almost" balanced binary search tree. This is accomplished by `avl.h`.

The input size of the `avl.h` algorithm is `n`, the result `Y` is the height of AVL Tree.

In my main function I use `default_random_engine` to get random number from 0 – `n` (`n` is my input size) and push this value by the function `insert` to get the AVL tree.

In the function `void insert(const Comparable& data, AVL_node*& t)`:

```
1       if (t == nullptr)
2       {
3         t = new AVL_node(data, nullptr, nullptr);
4       }
5       else if (data < t->data)
6       {
7         insert(data, t->left);
8         ...
9       }
10      else if (t->data < data)
11      {
12        insert(data, t->right);
13        ...
14      }
15      t->height = std::max(height(t->left), height(t->right)) + 1;
```

In function `insert`, we use recursion to find where we could insert the data in the AVL tree. If `data` is smaller than `t->data`, so `data` goes `insert(data, t->left)` in Line 7, so does for `insert(data, t->right)` in Line 12, until it finds that `t` is `nullptr`, we a new `AVL_node` and insert `data` here. When the `insert` function calls, it always runs Line 15, to get each height of the node `t` that roots the subtree.

After insert `data`, function will test the height-balanced of the node `t` that roots the subtree beginning with `data`'s parent until to the top that roots the AVL tree. if `t`'s height-balanced is equals to 2, it will run `RR`, `RL`, `LL` and `LR` functions in lines 8, and 13 which I don't quote in the sample to make binary search tree "almost" balance.

For function `rotateRR`:

```
void rotateRR(AVL_node*& p)
  {
    AVL_node* orig_right = p->right;
    p->right = orig_right->left;
    orig_right->left = p;
    p->height = std::max(height(p->right), height(p->left)) + 1;
    orig_right->height = std::max(height(orig_right->right), p->height) + 1;
```

```
      p = orig_right;
    }
```

It runs when the height-balanced of node `p` is equal to 2, and `data` is larger than `P->right->data`. In other words, when the new unbalancing node is at right of the AVL tree, we will use function RR that is rebalanced with rotating left to make it into a new balance (height-balanced < 2).

So does function `LL` do, it is rebalanced with rotating right to make it into a new balance. Also, it always runs when the new unbalancing node is at left of the AVL tree.

For function `RL`:

```
void rotateRL(AVL_node*& p)
  {
    AVL_node *temp_p = p;
    rotateLL(temp_p->right);
    rotateRR(p);
  }
```

Function `rotateRL` runs function `LL` first at the right of `p` , and then it runs function `RR` to make the whole tree being balance again.

The situation of function `LR` is the opposite. It runs function `RR` at the left of `p` first and then it is function `LL`, so it could make the tree into balance again.

After running all rotate functions, the height-balanced of all nodes in the AVL tree are less than 2. In other words, every root and its leave are almost evenly distributed.

After runs with different input size:

```
for n in $(seq 100 10 10000)
do
    ./program $n
    ./program $n
    ./program $n
done 2> results.dat
```

So, we could get: `n` is input size, `Y` is the height of AVL tree.

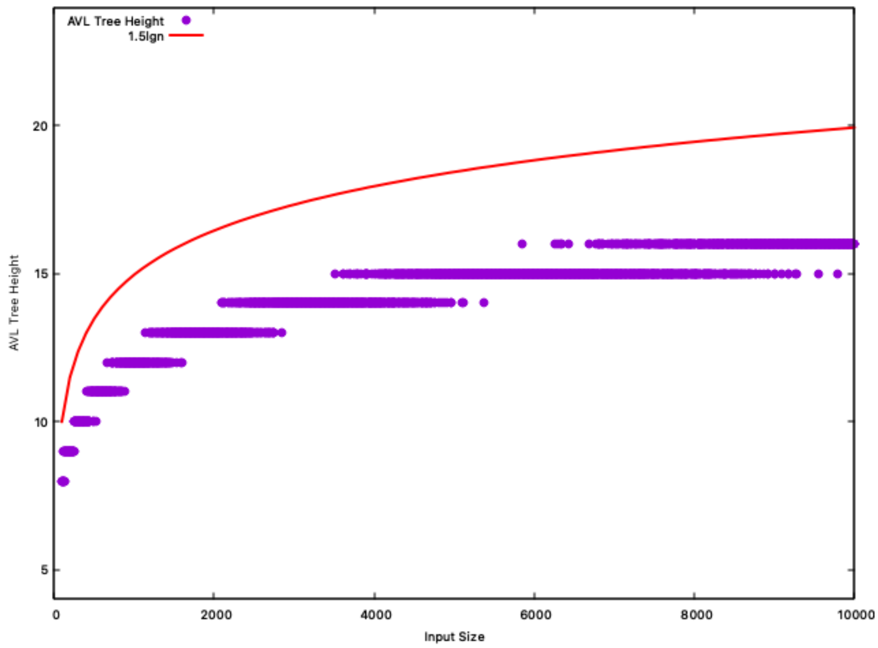
n	Y
1	1
2	2
3	2
4	3
5	3
100	8
1000	12
10000	16

Table 1: the height of AVL tree and input size relationship

Thus,

$$\begin{aligned} Y &\leq 1.5 \lg n \\ &\in \Omega(\lg n) \end{aligned}$$

in order to generate a set of points. The resulting data were plotted, giving the following. Also plotted on the same axes are the scaled standard functions $1.5 \lg n$ which illustrate above the `AVL Tree Height` is worst case.



We see that the plot confirms the theoretical analysis above. It is same as we told in class.