Motives of Balanced Trees

<table>
<thead>
<tr>
<th># of entries</th>
<th>Worst-case search steps</th>
<th>complete binary tree</th>
<th>skewed binary tree</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>4</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>63</td>
<td>6</td>
<td>63</td>
<td></td>
</tr>
<tr>
<td>1023</td>
<td>10</td>
<td>1023</td>
<td></td>
</tr>
<tr>
<td>65535</td>
<td>16</td>
<td>65535</td>
<td></td>
</tr>
<tr>
<td>1,048,575</td>
<td>20</td>
<td>1048575</td>
<td></td>
</tr>
<tr>
<td>1,073,741,823</td>
<td>30</td>
<td>1073741823</td>
<td></td>
</tr>
</tbody>
</table>

Analysis

- $\log_x y = a$ number $r$ such that $x^r = y$
  - $\log_2 1024 = 10$
  - $\log_2 65536 = 16$
  - $\log_2 1048576 = 20$
  - $\log_2 1073741824 = 30$
- $\lfloor x \rfloor = a$ integer $a$ such that $a \leq x$
  - $\lfloor 3.1416 \rfloor = 3$
  - $\lfloor 3.1416 \rfloor = 4$
  - $\lfloor 5 \rfloor = 5 = \lfloor 5 \rfloor$
If a binary search tree of $N$ nodes happens to be complete, then a search in the tree requires at most

$$\lceil \log_2 N \rceil + 1$$

steps.

- Big-$O$ notation: We say $f(x) = O(g(x))$ if $f(x)$ is bounded by $c \cdot g(x)$, where $c$ is a constant, for sufficiently large $x$.

  - $x^2 + 11x - 30 = O(x^2)$ (consider $c = 2$, $x \geq 6$)
  - $x^{100} + x^{50} + 2^x = O(2^x)$

- If a BST happens to be complete, then a search in the tree requires $O(\log_2 N)$ steps.

- In a linked list of $N$ nodes, a search requires $O(N)$ steps.

---

**AVL Trees**

- A balanced binary search tree structure proposed by Adelson, Velksii and Landis in 1962.

- In an AVL tree of $N$ nodes, every searching, deletion or insertion operation requires only $O(\log_2 N)$ steps.

- AVL tree searching is exactly the same as that of the binary search tree.
**Definition**

- An empty tree is balanced.
- If $T$ is a nonempty binary tree with $T_L$ and $T_R$ as its left right subtrees, then $T$ is balanced if and only if
  1. $T_L$ and $T_R$ are balanced.
  2. The difference between the heights of $T_L$ and $T_R$ is at most one.

**Examples**
Insertion

- The first step is to insert the new item $X$ to the tree using the rules of BST insertion.
- If the resultant tree is balanced, we are done.
- Otherwise, we must reorganize the tree to restore balance, as follows: From $X$ to the root, let $A$ be the first ancestor of $X$ whose corresponding subtree is no longer balanced due to the insertion of $X$. We consider four cases.

**LL.** $X$ is in the left-left subtree of $A$.

**RR.** $X$ is in the right-right subtree of $A$.

**LR.** $X$ is in the left-right subtree of $A$.

**RL.** $X$ is in the right-left subtree of $A$.

**Case LL**

This is called a rotate-right operation.

When this case occurs, the heights of $A_R$, $B_L$ and $B_R$ (before the insertion of $X$) must be identical. Why?

This property suggests that the above figure depicts the only configuration of Case LL.
This is called a rotate-left operation.

Can you convince yourself that this is the only configuration of Case RR?

Case RL

We further distinguish 3 cases.

- **RL-R.** $X$ is in the right-left-right subtree of $A$.
- **RL-L.** $X$ is in the right-left-left subtree of $A$.
- **RL-X.** $X$ is the right-left grand child of $A$. 
Claim: \( \text{height}(A_L) = \text{height}(B_R) \) and \( \text{height}(C_L) = \text{height}(C_R) \).
Further, \( A_L \) and \( B_R \) are exactly one level higher than \( C_L \) and \( C_R \).
Why?

That is, the above figure depicts the only configuration of RL-R.

---

Double Rotation

The RL-R transformation can be performed by first rotating right the “B-tree” and second rotating left the “A-tree.” This is called a double rotation.
As seen, RL-R and RL-L are the same transformation.
Can you convince yourself this is the only configuration of RL-L?

This transformation can also be performed by first rotating right the $B$-tree and second rotating left the $A$-tree.

**Conclusion:** The three RL cases are resolved by identical transformations.
However, we will realize when discussing implementations that the three cases differ in some book-keeping activities.
Case LR

We show only the LR-L case here.

Again, this is a double rotation: first rotating left the $B$-tree and second rotating right the $A$-tree.

The same transformation applies to LR-R and LR-X.

Examples

30  Insert 50  30  Insert 100  30  RR  A=30  50  50  30  100  50
Insert 20
50
30
20
10

Insert 10
50
30
20
10

LL
A=30
50
20
10
30

Insert 25
50
20
25
10
30

LR
A=50
30
20
25
10
50

Insert 28
30
20
23
10
25
28

Insert 23
30
20
28
10
25
23

Insert 24

RL
A=20

Insert 29
An Example of AVL Deletion

[Diagram of an AVL tree with nodes labeled from a to m, showing the deletion process.]
Discussion

- In handling deletions, we still use rotations and double rotations as tree re-organization tools.
- For insertions, a rotation or double rotation is performed at node $A$ and we are done.
- However, a deletion can trigger a sequence of rotations, from $X$ all the way up to the root.
- Details of AVL deletion are left as a project.

Implementation

Each node of an AVL tree is a binary tree node plus a balance factor, which is of value left-higher, equal-height, or right-higher.

```cpp
generic
enum Balance_factor { left_higher, equal_height, right_higher };
template<class ItemType>
struct AVL_node {
    ItemType data;
    AVL_node* left;
    AVL_node* right;
    Balance_factor balance;
};
```
Balance Factors

- equal height
- right heigher
- left heigher

The AVL_tree Class

template<ItemType>
class AVL_tree {
private:
    AVL_node* root;
    // auxiliary functions should be declared here

public:
    AVL_tree () {root = NULL;}
    bool insert (const ItemType k);
    bool remove (const ItemType k);
    // other functions, such as empty(), height(),
    // search(), inorder(), postorder(), and preorder().
};
Implementation of AVL Insertion

First, the public insert() function simply calls upon an auxiliary function, avl_insert(), to get the job done.

```cpp
template<class ItemType>
bool AVL_tree<ItemType>::insert (const ItemType& new_data)
{
    bool taller;
    return avl_insert (root, new_data, taller);
}
```

```cpp
template<class ItemType>
bool AVL_tree<ItemType>::avl_insert (AVL_node* r,
const ItemType& new_data, bool& taller)
{
    if (r==NULL) {
        r = new AVL_node<ItemType> (new_data);
        taller = true;
        return true;
    }
    if (new_data == r->data) {
        taller = false;
        return false;
    }
    if (new_data < r->data) {
        if (avl_insert(r->left,new_data,taller) == false)
            return false;
    }
    if (new_data > r->data) {
        if (avl_insert(r->right,new_data,taller) == false)
            return false;
    }
    return true;
}
```
if (taller==true)
    switch (r->balance) {
    case equal_height:
        r->balance = left_higher;
        break;
    case left_higher:
        left_balance (r);
        taller = false;    // Why ?
        break;
    case right_higher:
        r->balance = ?
        taller = false;
        break
    } /* end of switch (r->balance) */
} /* end of if (new_data < r->data) */

// Let us try insertion in right subtree.
if (new_data > r->data) {
    /* end of if (new_data > r->data) */
} /* end of avl_insert */
Implementing Rotations

template<class T>
void AVL_tree<T>::rotate_left (AVL_node<T>*& r)
{
    Binary_node<T>* a=r;
    Binary_node<T>* b=a->right;
}

Routine rotate_right() is left as an exercise.

Implementing Re-balancing Functions

template<class ItemType>
void AVL_tree<ItemType>::right_balance
    (AVL_node<ItemType>*& a)
{
    AVL_node<ItemType>*& b = a->right;
    if (b->balance == right_higher) { // case RR
        rotate_left (a);
        a->balance = b->balance = equal_height;
        return;
    } // since b->balance cannot be equal_height, now
    // b->balance must be left_higher --- case RL
    AVL_node<ItemType>* c = b->left;

// Update balance factors
switch (c->balance)
  case equal_height:  // Case RL-X
    a->balance = b->balance = equal_height;  break;
  case left_higher:   // Case RL-L
    a->balance = equal_height;
    b->balance = right_higher;  break;
  case right_higher:  // Case RL-R
    a->balance = left_higher;
    b->balance = equal_height;  break;
} /* end of switch(c->balance) */
c->balance = equal_height;

rotate_right (b);
rotate_left (a);
return;

Routine left_balance() is left as an exercise.

---

**Visualizing the Recursive Structure**

Example 1:

Insert 29

```
  30
 /   \
23  29
```


Example 2:

Performance Analysis

- We would like to determine the height of an AVL tree, which in turn determines the worst case performance of searching, insertion, and deletion on that tree.

- Given a fixed number of nodes $n$ in the tree, the shape of the tree depends on the order of insertion.
  - In the best cases, the tree could be a full/binary tree, resulting the optimal $O(\log N)$ performance.

- To consider the worst-case performance, we may want to determine the maximum height that an AVL tree with $N$ nodes can have.

- Instead, we ask the question in another way: what is the minimum number of nodes an AVL tree of height $h$ can have.
Let $F_h$ be the minimum of nodes in an AVL tree of height $h$. We have

\[
F_0 = 0 \\
F_1 = 1 \\
F_h = F_{h-1} + F_{h-2} + 1
\]

It can be shown that

\[
F_h \approx \frac{1}{\sqrt{5}} \left( \frac{1 + \sqrt{5}}{2} \right)^{h+2}
\]

that is, $h \approx 1.44 \log_2 F_h$

- Put in another way, the maximum height of an AVL tree with $N$ nodes is approximately $1.44 \log_2 N$.

- For an AVL tree with one billion nodes, its worst-case tree height is less than $1.5*30 = 45$.

- This verifies our earlier claim that for an AVL tree of $N$ nodes every searching, deletion or insertion operation requires only $O(\log_2 N)$ steps.