AVL tree

Liu Xinyu [*]

January 5, 2013

1 Introduction

1.1 How to measure the balance of a tree?

Besides red-black tree, are there any other intuitive solutions of self-balancing binary search tree? In order to measure how balancing a binary search tree, one idea is to compare the height of the left sub-tree and right sub-tree. If they differs a lot, the tree isn't well balanced. Let's denote the difference height between two children as below

$$\delta(T) = |L| - |R| \tag{1}$$

Where $|T|$ means the height of tree T, and L, R denotes the left sub-tree and right sub-tree.

If $\delta(T) = 0$, The tree is definitely balanced. For example, a complete binary tree has $N = 2^h - 1$ nodes for height h. There is no empty branches unless the leafs. Another trivial case is empty tree. $\delta(\phi) = 0$. The less absolute value of $\delta(T)$ the more balancing the tree is.

We define $\delta(T)$ as the *balance factor* of a binary search tree.

2 Definition of AVL tree

An AVL tree is a special binary search tree, that all sub-trees satisfying the following criteria.

$$|\delta(T)| \leq 1 \tag{2}$$

The absolute value of balance factor is less than or equal to 1, which means there are only three valid values, -1, 0 and 1. Figure 1 shows an example AVL tree.

[*]**Liu Xinyu**
Email: liuxinyu95@gmail.com

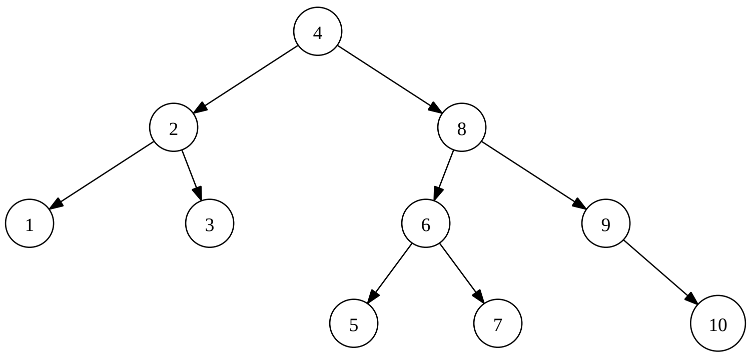

Figure 1: An example AVL tree

Why AVL tree can keep the tree balanced? In other words, Can this definition ensure the height of the tree as $O(\lg N)$ where N is the number of the nodes in the tree? Let's prove this fact.

For an AVL tree of height h, The number of nodes varies. It can have at most $2^h - 1$ nodes for a complete binary tree. We are interesting about how many nodes there are at least. Let's denote the minimum number of nodes for height h AVL tree as $N(h)$. It's obvious for the trivial cases as below.

- For empty tree, $h = 0$, $N(0) = 0$;
- For a singleton root, $h = 1$, $N(1) = 1$;

What's the situation for common case $N(h)$? Figure 2 shows an AVL tree T of height h. It contains three part, the root node, and two sub trees A, B. We have the following fact.

$$h = max(height(L), height(R)) + 1 \tag{3}$$

We immediately know that, there must be one child has height $h - 1$. Let's say $height(A) = h - 1$. According to the definition of AVL tree, we have. $|height(A) - height(B)| \leq 1$. This leads to the fact that the height of other tree B can't be lower than $h - 2$, So the total number of the nodes of T is the number of nodes in tree A, and B plus 1 (for the root node). We exclaim that.

$$N(h) = N(h-1) + N(h-2) + 1 \tag{4}$$

This recursion reminds us the famous Fibonacci series. Actually we can transform it to Fibonacci series by defining $N'(h) = N(h) + 1$. So equation 4 changes to.

$$N'(h) = N'(h-1) + N'(h-2) \tag{5}$$

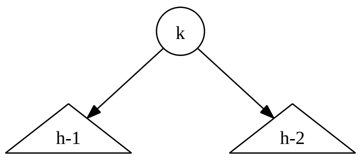

Figure 2: An AVL tree with height h, one of the sub-tree with height $h-1$, the other is $h-2$

Lemma 2.1. *Let $N(h)$ be the minimum number of nodes for an AVL tree with height h. and $N'(h) = N(h) + 1$, then*

$$N'(h) \geq \phi^h \tag{6}$$

Where $\phi = \frac{\sqrt{5}+1}{2}$ is the golden ratio.

Proof. For the trivial case, we have

- $h = 0$, $N'(0) = 1 \geq \phi^0 = 1$
- $h = 1$, $N'(1) = 2 \geq \phi^1 = 1.618...$

For the induction case, suppose $N'(h) \geq \phi^h$.

$$\begin{array}{rll} N'(h+1) & = N'(h) + N'(h-1) & \{Fibonacci\} \\ & \geq \phi^h + \phi^{h-1} & \\ & = \phi^{h-1}(\phi+1) & \{\phi + 1 = \phi^2 = \frac{\sqrt{5}+3}{2}\} \\ & = \phi^{h+1} & \end{array}$$

□

From Lemma 2.1, we immediately get

$$h \leq log_{\phi}(N+1) = log_{\phi}(2) \cdot \lg(N+1) \approx 1.44 \lg(N+1) \tag{7}$$

It tells that the height of AVL tree is proportion to $O(\lg N)$, which means that AVL tree is balanced.

During the basic mutable tree operations such as insertion and deletion, if the balance factor changes to any invalid value, some fixing has to be performed to resume $|\delta|$ within 1. Most implementations utilize tree rotations. In this chapter, we'll show the pattern matching solution which is inspired by Okasaki's red-black tree solution[2]. Because of this modify-fixing approach, AVL tree is also a kind of self-balancing binary search tree. For comparison purpose, we'll also show the procedural algorithms.

Of course we can compute the δ value recursively, another option is to store the balance factor inside each nodes, and update them when we modify the tree. The latter one avoid computing the same value every time.

Based on this idea, we can add one data field δ to the original binary search tree as the following C++ code example [1].

```
template <class T>
struct node{
  int delta;
  T key;
  node* left;
  node* right;
  node* parent;
};
```

In purely functional setting, some implementation use different constructor to store the δ information. for example in [1], there are 4 constructors, E, N, P, Z defined. E for empty tree, N for tree with negative 1 balance factor, P for tree with positive 1 balance factor and Z for zero case.

In this chapter, we'll explicitly store the balance factor inside the node.

```
data AVLTree a = Empty
               | Br (AVLTree a) a (AVLTree a) Int
```

The immutable operations, including looking up, finding the maximum and minimum elements are all same as the binary search tree. We'll skip them and focus on the mutable operations.

3 Insertion

Insert a new element to an AVL tree may violate the AVL tree property that the δ absolute value exceeds 1. To resume it, one option is to do the tree rotation according to the different insertion cases. Most implementation is based on this approach

Another way is to use the similar pattern matching method mentioned by Okasaki in his red-black tree implementation [2]. Inspired by this idea, it is possible to provide a simple and intuitive solution.

When insert a new key to the AVL tree, the balance factor of the root may changes in range $[-1, 1]$, and the height may increase at most by one, which we need recursively use this information to update the δ value in upper level nodes. We can define the result of the insertion algorithm as a pair of data $(T', \Delta H)$. Where T' is the new tree and ΔH is the increment of height. Let's denote function $first(pair)$ which can return the first element in a pair. We can modify the binary search tree insertion algorithm as the following to handle

[1]Some implementations store the height of a tree instead of δ as in [5]

AVL tree.

$$insert(T, k) = first(ins(T, k)) \tag{8}$$

where

$$ins(T, k) = \begin{cases} (node(\phi, k, \phi, 0), 1) & : \quad T = \phi \\ (tree(ins(L, k), Key, (R, 0)), \Delta) & : \quad k < Key \\ (tree((L, 0), Key, ins(R, k)), \Delta) & : \quad otherwise \end{cases} \tag{9}$$

L, R, Key, Δ represent the left child, right child, the key and the balance factor of a tree.

$$\begin{array}{l} L = left(T) \\ R = right(T) \\ Key = key(T) \\ \Delta = \delta(T) \end{array}$$

When we insert a new key k to a AVL tree T, if the tree is empty, we just need create a leaf node with k, set the balance factor as 0, and the height is increased by one. This is the trivial case. Function $node()$ is defined to build a tree by taking a left sub-tree, a right sub-tree, a key and a balance factor.

If T isn't empty, we need compare the Key with k. If k is less than the key, we recursively insert it to the left child, otherwise we insert it into the right child.

As we defined above, the result of the recursive insertion is a pair like $(L', \Delta H_l)$, we need do balancing adjustment as well as updating the increment of height. Function $tree()$ is defined to dealing with this task. It takes 4 parameters as $(L', \Delta H_l)$, Key, $(R', \Delta H_r)$, and Δ. The result of this function is defined as $(T', \Delta H)$, where T' is the new tree after adjustment, and ΔH is the new increment of height which is defined as

$$\Delta H = |T'| - |T| \tag{10}$$

This can be further detailed deduced in 4 cases.

$$\begin{aligned} \Delta H &= |T'| - |T| \\ &= 1 + max(|R'|, |L'|) - (1 + max(|R|, |L|)) \\ &= max(|R'|, |L'|) - max(|R|, |L|) \\ &= \begin{cases} \Delta H_r & : \quad \Delta \geq 0 \wedge \Delta' \geq 0 \\ \Delta + \Delta H_r & : \quad \Delta \leq 0 \wedge \Delta' \geq 0 \\ \Delta H_l - \Delta & : \quad \Delta \geq 0 \wedge \Delta' \leq 0 \\ \Delta H_l & : \quad otherwise \end{cases} \end{aligned} \tag{11}$$

To prove this equation, note the fact that the height can't increase both in left and right with only one insertion.

These 4 cases can be explained from the definition of balance factor definition that it equal to the difference from the right sub tree and left sub tree.

- If $\Delta \geq 0$ and $\Delta' \geq 0$, it means that the height of right sub tree isn't less than the height of left sub tree both before insertion and after insertion. In this case, the increment in height of the tree is only 'contributed' from the right sub tree, which is ΔH_r.

- If $\Delta \leq 0$, which means the height of left sub tree isn't less than the height of right sub tree before, and it becomes $\Delta' \geq 0$, which means that the height of right sub tree increases due to insertion, and the left side keeps same ($|L'| = |L|$). So the increment in height is

$$\begin{array}{rll} \Delta H & = max(|R'|, |L'|) - max(|R|, |L|) & \{\Delta \leq 0 \land \Delta' \geq 0\} \\ & = |R'| - |L'| & \{|L| = |L'|\} \\ & = |R| + \Delta H_r - |L| & \\ & = \Delta + \Delta H_r & \end{array}$$

- For the case $\Delta \geq 0 \land \Delta' \leq 0$, Similar as the second one, we can get.

$$\begin{array}{rll} \Delta H & = max(|R'|, |L'|) - max(|R|, |L|) & \{\Delta \geq 0 \land \Delta' \leq 0\} \\ & = |L'| - |R| & \\ & = |L| + \Delta H_l - |R| & \\ & = \Delta H_l - \Delta & \end{array}$$

- For the last case, the both Δ and Δ' is no bigger than zero, which means the height left sub tree is always greater than or equal to the right sub tree, so the increment in height is only 'contributed' from the right sub tree, which is ΔH_l.

The next problem in front of us is how to determine the new balancing factor value Δ' before performing balancing adjustment. According to the definition of AVL tree, the balancing factor is the height of right sub tree minus the height of right sub tree. We have the following facts.

$$\begin{array}{rl} \Delta' & = |R'| - |L'| \\ & = |R| + \Delta H_r - (|L| + \Delta H_l) \\ & = |R| - |L| + \Delta H_r - \Delta H_l \\ & = \Delta + \Delta H_r - \Delta H_l \end{array} \tag{12}$$

With all these changes in height and balancing factor get clear, it's possible to define the $tree()$ function mentioned in (9).

$$tree((L', \Delta H_l), Key, (R', \Delta H_r), \Delta) = balance(node(L', Key, R', \Delta'), \Delta H) \tag{13}$$

Before we moving into details of balancing adjustment, let's translate the above equations to real programs in Haskell.

First is the insert function.

```
insert::(Ord a)⇒AVLTree a → a → AVLTree a
insert t x = fst $ ins t where
    ins Empty = (Br Empty x Empty 0, 1)
    ins (Br l k r d)
        | x < k     = tree (ins l) k (r, 0) d
        | x == k    = (Br l k r d, 0)
        | otherwise = tree (l, 0) k (ins r) d
```

Here we also handle the case that inserting a duplicated key (which means the key has already existed.) as just overwriting.

```
tree::(AVLTree a, Int) → a → (AVLTree a, Int) → Int → (AVLTree a, Int)
tree (l, dl) k (r, dr) d = balance (Br l k r d', delta) where
    d' = d + dr - dl
    delta = deltaH d d' dl dr
```

And the definition of height increment is as below.

```
deltaH :: Int → Int → Int → Int → Int
deltaH d d' dl dr
       | d ≥0 && d' ≥0 = dr
       | d ≤0 && d' ≥0 = d+dr
       | d ≥0 && d' ≤0 = dl - d
       | otherwise = dl
```

3.1 Balancing adjustment

As the pattern matching approach is adopted in doing re-balancing. We need consider what kind of patterns violate the AVL tree property.

Figure 3 shows the 4 cases which need fix. For all these 4 cases the balancing factors are either -2, or +2 which exceed the range of $[-1, 1]$. After balancing adjustment, this factor turns to be 0, which means the height of left sub tree is equal to the right sub tree.

We call these four cases left-left lean, right-right lean, right-left lean, and left-right lean cases in clock-wise direction from top-left. We denote the balancing factor before fixing as $\delta(x), \delta(y)$, and $\delta(z)$, while after fixing, they changes to $\delta'(x), \delta'(y)$, and $\delta'(z)$ respectively.

We'll next prove that, after fixing, we have $\delta(y) = 0$ for all four cases, and we'll provide the result values of $\delta'(x)$ and $\delta'(z)$.

Left-left lean case

As the structure of sub tree x doesn't change due to fixing, we immediately get $\delta'(x) = \delta(x)$.

Since $\delta(y) = -1$ and $\delta(z) = -2$, we have

$$\begin{array}{l} \delta(y) = |C| - |x| = -1 \Rightarrow |C| = |x| - 1 \\ \delta(z) = |D| - |y| = -2 \Rightarrow |D| = |y| - 2 \end{array} \quad (14)$$

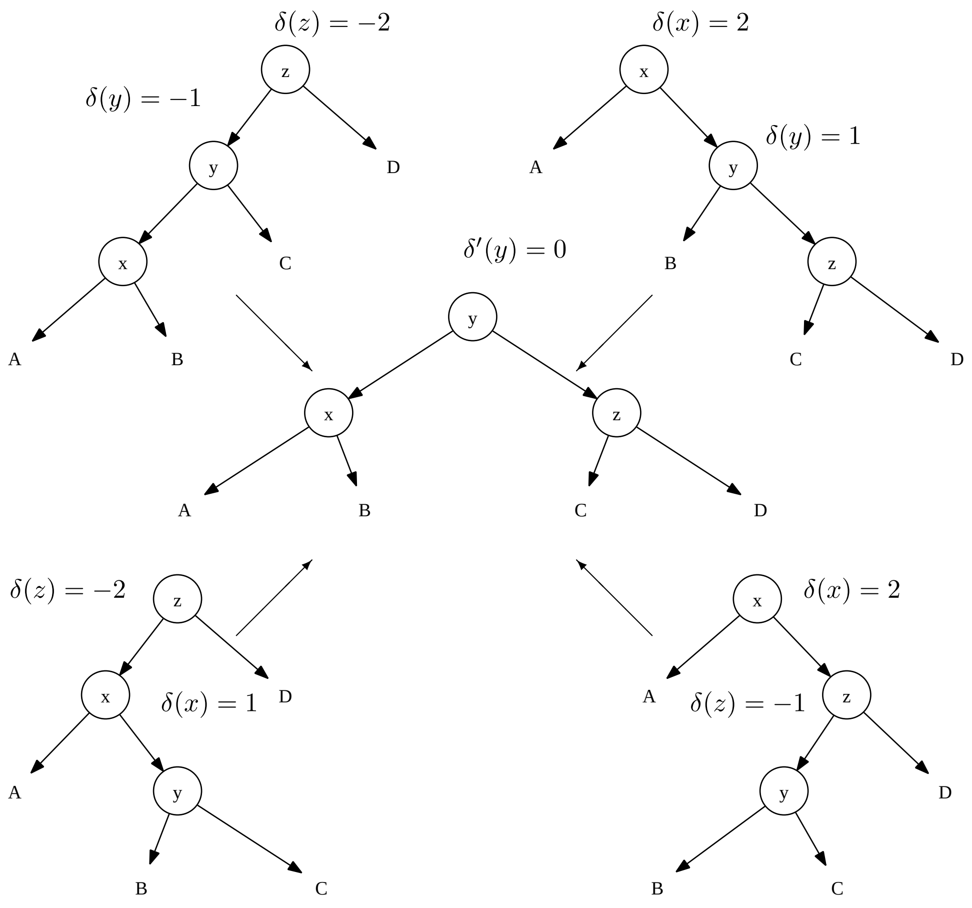

Figure 3: 4 cases for balancing a AVL tree after insertion

After fixing.

$$\begin{array}{ll} \delta'(z) & = |D| - |C| \quad \{From(14)\} \\ & = |y| - 2 - (|x| - 1) \\ & = |y| - |x| - 1 \quad \{x \text{ is child of } y \Rightarrow |y| - |x| = 1\} \\ & = 0 \end{array} \tag{15}$$

For $\delta'(y)$, we have the following fact after fixing.

$$\begin{array}{ll} \delta'(y) & = |z| - |x| \\ & = 1 + max(|C|, |D|) - |x| \quad \{\text{By (15), we have} |C| = |D|\} \\ & = 1 + |C| - |x| \quad \{\text{By (14)}\} \\ & = 1 + |x| - 1 - |x| \\ & = 0 \end{array} \tag{16}$$

Summarize the above results, the left-left lean case adjust the balancing factors as the following.

$$\begin{array}{l} \delta'(x) = \delta(x) \\ \delta'(y) = 0 \\ \delta'(z) = 0 \end{array} \tag{17}$$

Right-right lean case

Since right-right case is symmetric to left-left case, we can easily achieve the result balancing factors as

$$\begin{array}{l} \delta'(x) = 0 \\ \delta'(y) = 0 \\ \delta'(z) = \delta(z) \end{array} \tag{18}$$

Right-left lean case

First let's consider $\delta'(x)$. After balance fixing, we have.

$$\delta'(x) = |B| - |A| \tag{19}$$

Before fixing, if we calculate the height of z, we can get.

$$\begin{array}{ll} |z| & = 1 + max(|y|, |D|) \quad \{\delta(z) = -1 \Rightarrow |y| > |D|\} \\ & = 1 + |y| \\ & = 2 + max(|B|, |C|) \end{array} \tag{20}$$

While since $\delta(x) = 2$, we can deduce that.

$$\begin{array}{ll} \delta(x) = 2 & \Rightarrow |z| - |A| = 2 \quad \{\text{By (20)}\} \\ & \Rightarrow 2 + max(|B|, |C|) - |A| = 2 \\ & \Rightarrow max(|B|, |C|) - |A| = 0 \end{array} \tag{21}$$

If $\delta(y) = 1$, which means $|C| - |B| = 1$, it means

$$max(|B|, |C|) = |C| = |B| + 1 \tag{22}$$

Take this into (21) yields

$$\begin{aligned} & |B| + 1 - |A| = 0 \Rightarrow |B| - |A| = -1 \quad \{\text{By } (19)\ \} \\ & \Rightarrow \delta'(x) = -1 \end{aligned} \tag{23}$$

If $\delta(y) \neq 1$, it means $max(|B|, |C|) = |B|$, taking this into (21), yields.

$$\begin{aligned} & |B| - |A| = 0 \quad \{\text{By } (19)\} \\ & \Rightarrow \delta'(x) = 0 \end{aligned} \tag{24}$$

Summarize these 2 cases, we get relationship of $\delta'(x)$ and $\delta(y)$ as the following.

$$\delta'(x) = \begin{cases} -1 & : \quad \delta(y) = 1 \\ \ \ 0 & : \quad otherwise \end{cases} \tag{25}$$

For $\delta'(z)$ according to definition, it is equal to.

$$\begin{aligned} \delta'(z) \quad &= |D| - |C| && \{\delta(z) = -1 = |D| - |y|\} \\ &= |y| - |C| - 1 && \{|y| = 1 + max(|B|, |C|)\} \\ &= max(|B|, |C|) - |C| \end{aligned} \tag{26}$$

If $\delta(y) = -1$, then we have $|C| - |B| = -1$, so $max(|B|, |C|) = |B| = |C| + 1$. Takes this into (26), we get $\delta'(z) = 1$.

If $\delta(y) \neq -1$, then $max(|B|, |C|) = |C|$, we get $\delta'(z) = 0$.

Combined these two cases, the relationship between $\delta'(z)$ and $\delta(y)$ is as below.

$$\delta'(z) = \begin{cases} 1 & : \quad \delta(y) = -1 \\ 0 & : \quad otherwise \end{cases} \tag{27}$$

Finally, for $\delta'(y)$, we deduce it like below.

$$\begin{aligned} \delta'(y) \quad &= |z| - |x| \\ &= max(|C|, |D|) - max(|A|, |B|) \end{aligned} \tag{28}$$

There are three cases.

- If $\delta(y) = 0$, it means $|B| = |C|$, and according to (25) and (27), we have $\delta'(x) = 0 \Rightarrow |A| = |B|$, and $\delta'(z) = 0 \Rightarrow |C| = |D|$, these lead to $\delta'(y) = 0$.

- If $\delta(y) = 1$, From (27), we have $\delta'(z) = 0 \Rightarrow |C| = |D|$.

$$\begin{aligned} \delta'(y) \quad &= max(|C|, |D|) - max(|A|, |B|) && \{|C| = |D|\} \\ &= |C| - max(|A|, |B|) && \{\text{From } (25)\text{: } \delta'(x) = -1 \Rightarrow |B| - |A| = -1\} \\ &= |C| - (|B| + 1) && \{\delta(y) = 1 \Rightarrow |C| - |B| = 1\} \\ &= 0 \end{aligned}$$

- If $\delta(y) = -1$, From (25), we have $\delta'(x) = 0 \Rightarrow |A| = |B|$.

$$\begin{array}{rll}
\delta'(y) & = max(|C|,|D|) - max(|A|,|B|) & \{|A| = |B|\} \\
 & = max(|C|,|D|) - |B| & \{\text{From (27): } |D| - |C| = 1\} \\
 & = |C| + 1 - |B| & \{\delta(y) = -1 \Rightarrow |C| - |B| = -1\} \\
 & = 0 &
\end{array}$$

Both three cases lead to the same result that $\delta'(y) = 0$.

Collect all the above results, we get the new balancing factors after fixing as the following.

$$\begin{array}{l}
\delta'(x) = \left\{ \begin{array}{rcl} -1 & : & \delta(y) = 1 \\ 0 & : & otherwise \end{array} \right. \\
\delta'(y) = 0 \\
\delta'(z) = \left\{ \begin{array}{rcl} 1 & : & \delta(y) = -1 \\ 0 & : & otherwise \end{array} \right.
\end{array} \tag{29}$$

Left-right lean case

Left-right lean case is symmetric to the Right-left lean case. By using the similar deduction, we can find the new balancing factors are identical to the result in (29).

3.2 Pattern Matching

All the problems have been solved and it's time to define the final pattern matching fixing function.

$$balance(T, \Delta H) = \left\{ \begin{array}{rcl}
(node(node(A,x,B,\delta(x)),y,node(C,z,D,0),0),0) & : & P_{ll}(T) \\
(node(node(A,x,B,0),y,node(C,z,D,\delta(z)),0),0) & : & P_{rr}(T) \\
(node(node(A,x,B,\delta'(x)),y,node(C,z,D,\delta'(z)),0),0) & : & P_{rl}(T) \vee P_{lr}(T) \\
(T, \Delta H) & : & otherwise
\end{array} \right. \tag{30}$$

Where $P_{ll}(T)$ means the pattern of tree T is left-left lean respectively. $\delta'(x)$ and $delta'(z)$ are defined in (29). The four patterns are tested as below.

$$\begin{array}{l}
P_{ll}(T) = node(node(node(A,x,B,\delta(x)),y,C,-1),z,D,-2) \\
P_{rr}(T) = node(A,x,node(B,y,node(C,z,D,\delta(z)),1),2) \\
P_{rl}(T) = node(node(A,x,node(B,y,C,\delta(y)),1),z,D,-2) \\
P_{lr}(T) = node(A,x,node(node(B,y,C,\delta(y)),z,D,-1),2)
\end{array} \tag{31}$$

Translating the above function definition to Haskell yields a simple and intuitive program.

```
balance :: (AVLTree a, Int) → (AVLTree a, Int)
balance (Br (Br (Br a x b dx) y c (-1)) z d (-2), _) =
       (Br (Br a x b dx) y (Br c z d 0) 0, 0)
```

```
balance (Br a x (Br b y (Br c z d dz)    1)    2, _) =
        (Br (Br a x b 0) y (Br c z d dz) 0, 0)
balance (Br (Br a x (Br b y c dy)    1) z d (-2), _) =
        (Br (Br a x b dx') y (Br c z d dz') 0, 0) where
    dx' = if dy ==  1 then -1 else 0
    dz' = if dy == -1 then  1 else 0
balance (Br a x (Br (Br b y c dy) z d (-1))    2, _) =
        (Br (Br a x b dx') y (Br c z d dz') 0, 0) where
    dx' = if dy ==  1 then -1 else 0
    dz' = if dy == -1 then  1 else 0
balance (t, d) = (t, d)
```

The insertion algorithm takes time proportion to the height of the tree, and according to the result we proved above, its performance is $O(\lg N)$ where N is the number of elements stored in the AVL tree.

3.2.1 Verification

One can easily create a function to verify a tree is AVL tree. Actually we need verify two things, first, it's a binary search tree; second, it satisfies AVL tree property.

We left the first verification problem as an exercise to the reader.

In order to test if a binary tree satisfies AVL tree property, we can test the difference in height between its two children, and recursively test that both children conform to AVL property until we arrive at an empty leaf.

$$avl?(T) = \left\{ \begin{array}{rcl} True & : & T = \Phi \\ avl?(L) \land avl?(R) \land ||R| - |L|| \leq 1 & : & otherwise \end{array} \right. \tag{32}$$

And the height of a AVL tree can also be calculate from the definition.

$$|T| = \left\{ \begin{array}{rcl} 0 & : & T = \Phi \\ 1 + max(|R|, |L|) & : & otherwise \end{array} \right. \tag{33}$$

The corresponding Haskell program is given as the following.

```
isAVL :: (AVLTree a) → Bool
isAVL Empty = True
isAVL (Br l _ r d) = and [isAVL l, isAVL r, abs (height r - height l) ≤ 1]

height :: (AVLTree a) → Int
height Empty = 0
height (Br l _ r _) = 1 + max (height l) (height r)
```

Exercise 1

Write a program to verify a binary tree is a binary search tree in your favorite programming language. If you choose to use an imperative language, please consider realize this program without recursion.

4 Deletion

As we mentioned before, deletion doesn't make significant sense in purely functional settings. As the tree is read only, it's typically performs frequently looking up after build.

Even if we implement deletion, it's actually re-building the tree as we presented in chapter of red-black tree. We left the deletion of AVL tree as an exercise to the reader.

Exercise 2

- Take red-black tree deletion algorithm as an example, write the AVL tree deletion program in purely functional approach in your favorite programming language.
- Write the deletion algorithm in imperative approach in your favorite programming language.

5 Imperative AVL tree algorithm ⋆

We almost finished all the content in this chapter about AVL tree. However, it necessary to show the traditional insert-and-rotate approach as the comparator to pattern matching algorithm.

Similar as the imperative red-black tree algorithm, the strategy is first to do the insertion as same as for binary search tree, then fix the balance problem by rotation and return the final result.

```
function INSERT($T, k$)
    $root \leftarrow T$
    $x \leftarrow$ CREATE-LEAF($k$)
    $\delta(x) \leftarrow 0$
    $parent \leftarrow NIL$
    while $T \neq NIL$ do
        $parent \leftarrow T$
        if $k <$ KEY($T$) then
            $T \leftarrow$ LEFT($T$)
        else
            $T \leftarrow$ RIGHT($T$)
    PARENT($x$) $\leftarrow parent$
    if $parent = NIL$ then                              ▷ tree $T$ is empty
        return $x$
    else if $k <$ KEY($parent$) then
        LEFT($parent$) $\leftarrow x$
    else
        RIGHT($parent$) $\leftarrow x$
```

```
19:     return AVL-INSERT-FIX($root, x$)
```

Note that after insertion, the height of the tree may increase, so that the balancing factor δ may also change, insert on right side will increase δ by 1, while insert on left side will decrease it. By the end of this algorithm, we need perform bottom-up fixing from node x towards root.

We can translate the pseudo code to real programming language, such as Python [2].

```
def avl_insert(t, key):
    root = t
    x = Node(key)
    parent = None
    while(t):
        parent = t
        if(key < t.key):
            t = t.left
        else:
            t = t.right
    if parent is None: #tree is empty
        root = x
    elif key < parent.key:
        parent.set_left(x)
    else:
        parent.set_right(x)
    return avl_insert_fix(root, x)
```

This is a top-down algorithm search the tree from root down to the proper position and insert the new key as a leaf. By the end of this algorithm, it calls fixing procedure, by passing the root and the new node inserted.

Note that we reuse the same methods of set_left() and set_right() as we defined in chapter of red-black tree.

In order to resume the AVL tree balance property by fixing, we first determine if the new node is inserted on left hand or right hand. If it is on left, the balancing factor δ decreases, otherwise it increases. If we denote the new value as δ', there are 3 cases of the relationship between δ and δ'.

- If $|\delta| = 1$ and $|\delta'| = 0$, this means adding the new node makes the tree perfectly balanced, the height of the parent node doesn't change, the algorithm can be terminated.
- If $|\delta| = 0$ and $|\delta'| = 1$, it means that either the height of left sub tree or right sub tree increases, we need go on check the upper level of the tree.
- If $|\delta| = 1$ and $|\delta'| = 2$, it means the AVL tree property is violated due to the new insertion. We need perform rotation to fix it.

```
1: function AVL-INSERT-FIX($T, x$)
```

[2] C and C++ source code are available along with this book

```
2:  while $\textsc{Parent}(x) \neq NIL$ do
3:      $\delta \leftarrow \delta(\textsc{Parent}(x))$
4:      if $x = \textsc{Left}(\textsc{Parent}(x))$ then
5:          $\delta' \leftarrow \delta - 1$
6:      else
7:          $\delta' \leftarrow \delta + 1$
8:      $\delta(\textsc{Parent}(x)) \leftarrow \delta'$
9:      $P \leftarrow \textsc{Parent}(x)$
10:     $L \leftarrow \textsc{Left}(x)$
11:     $R \leftarrow \textsc{Right}(x)$
12:     if $|\delta| = 1$ and $|\delta'| = 0$ then        ▷ Height doesn't change, terminates.
13:         return $T$
14:     else if $|\delta| = 0$ and $|\delta'| = 1$ then          ▷ Go on bottom-up updating.
15:         $x \leftarrow P$
16:     else if $|\delta| = 1$ and $|\delta'| = 2$ then
17:         if $\delta' = 2$ then
18:             if $\delta(R) = 1$ then                                ▷ Right-right case
19:                 $\delta(P) \leftarrow 0$                                     ▷ By (18)
20:                 $\delta(R) \leftarrow 0$
21:                 $T \leftarrow \textsc{Left-Rotate}(T, P)$
22:             if $\delta(R) = -1$ then                                ▷ Right-left case
23:                 $\delta_y \leftarrow \delta(\textsc{Left}(R))$                            ▷ By (29)
24:                 if $\delta_y = 1$ then
25:                     $\delta(P) \leftarrow -1$
26:                 else
27:                     $\delta(P) \leftarrow 0$
28:                 $\delta(\textsc{Left}(R)) \leftarrow 0$
29:                 if $\delta_y = -1$ then
30:                     $\delta(R) \leftarrow 1$
31:                 else
32:                     $\delta(R) \leftarrow 0$
33:                 $T \leftarrow \textsc{Right-Rotate}(T, R)$
34:                 $T \leftarrow \textsc{Left-Rotate}(T, P)$
35:         if $\delta' = -2$ then
36:             if $\delta(L) = -1$ then                                 ▷ Left-left case
37:                 $\delta(P) \leftarrow 0$
38:                 $\delta(L) \leftarrow 0$
39:                 $\textsc{Right-Rotate}(T, P)$
40:             else                                                    ▷ Left-Right case
41:                 $\delta_y \leftarrow \delta(\textsc{Right}(L))$
42:                 if $\delta_y = 1$ then
43:                     $\delta(L) \leftarrow -1$
44:                 else
45:                     $\delta(L) \leftarrow 0$
```

```
46:                 $\delta(\textsc{Right}(L)) \leftarrow 0$
47:                 if $\delta_y = -1$ then
48:                     $\delta(P) \leftarrow 1$
49:                 else
50:                     $\delta(P) \leftarrow 0$
51:                 Left-Rotate($T, L$)
52:                 Right-Rotate($T, P$)
53:             break
54:     return $T$
```

Here we reuse the rotation algorithms mentioned in red-black tree chapter. Rotation operation doesn't update balancing factor δ at all, However, since rotation changes (actually improves) the balance situation we should update these factors. Here we refer the results from above section. Among the four cases, right-right case and left-left case only need one rotation, while right-left case and left-right case need two rotations.

The relative python program is shown as the following.

```python
def avl_insert_fix(t, x):
    while x.parent is not None:
        d2 = d1 = x.parent.delta
        if x == x.parent.left:
            d2 = d2 - 1
        else:
            d2 = d2 + 1
        x.parent.delta = d2
        (p, l, r) = (x.parent, x.parent.left, x.parent.right)
        if abs(d1) == 1 and abs(d2) == 0:
            return t
        elif abs(d1) == 0 and abs(d2) == 1:
            x = x.parent
        elif abs(d1)==1 and abs(d2) == 2:
            if d2 == 2:
                if r.delta == 1:  # Right-right case
                    p.delta = 0
                    r.delta = 0
                    t = left_rotate(t, p)
                if r.delta == -1: # Right-Left case
                    dy = r.left.delta
                    if dy == 1:
                        p.delta = -1
                    else:
                        p.delta = 0
                    r.left.delta = 0
                    if dy == -1:
                        r.delta = 1
                    else:
                        r.delta = 0
                    t = right_rotate(t, r)
                    t = left_rotate(t, p)
            if d2 == -2:
                if l.delta == -1: # Left-left case
                    p.delta = 0
                    l.delta = 0
                    t = right_rotate(t, p)
                if l.delta == 1: # Left-right case
                    dy = l.right.delta
                    if dy == 1:
                        l.delta = -1
                    else:
                        l.delta = 0
                    l.right.delta = 0
                    if dy == -1:
                        p.delta = 1
                    else:
                        p.delta = 0
                    t = left_rotate(t, l)
                    t = right_rotate(t, p)
            break
    return t
```

We skip the AVL tree deletion algorithm and left this as an exercise to the reader.

6 Chapter note

AVL tree was invented in 1962 by Adelson-Velskii and Landis[3], [4]. The name AVL tree comes from the two inventor's name. It's earlier than red-black tree.

It's very common to compare AVL tree and red-black tree, both are self-balancing binary search trees, and for all the major operations, they both consume $O(\lg N)$ time. From the result of (7), AVL tree is more rigidly balanced hence they are faster than red-black tree in looking up intensive applications [3]. However, red-black trees could perform better in frequently insertion and removal cases.

Many popular self-balancing binary search tree libraries are implemented on top of red-black tree such as STL etc. However, AVL tree provides an intuitive and effective solution to the balance problem as well.

After this chapter, we'll extend the tree data structure from storing data in node to storing information on edges, which leads to Trie and Patrica, etc. If we extend the number of children from two to more, we can get B-tree. These data structures will be introduced next.

References

[1] Data.Tree.AVL http://hackage.haskell.org/packages/archive/AvlTree/4.2/doc/html/Data-Tree-AVL.html

[2] Chris Okasaki. “FUNCTIONAL PEARLS Red-Black Trees in a Functional Setting”. J. Functional Programming. 1998

[3] Wikipedia. “AVL tree”. http://en.wikipedia.org/wiki/AVL_tree

[4] Guy Cousinear, Michel Mauny. “The Functional Approach to Programming”. Cambridge University Press; English Ed edition (October 29, 1998). ISBN-13: 978-0521576819

[5] Pavel Grafov. “Implementation of an AVL tree in Python”. http://github.com/pgrafov/python-avl-tree