16 AVL Trees

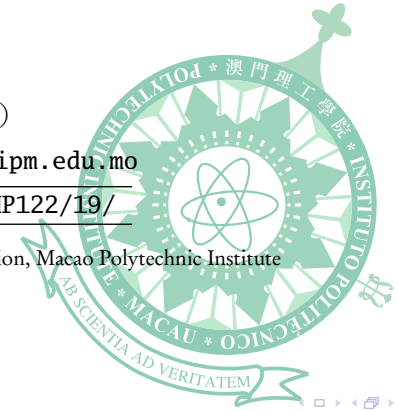
Instructor : Ke Wei（柯韋）

A319 ✆ Ext. 6452 ✉ wke@ipm.edu.mo

http://brouwer.ipm.edu.mo/COMP122/19/

Bachelor of Science in Computing, School of Public Administration, Macao Polytechnic Institute

March 22, 2019



Outline

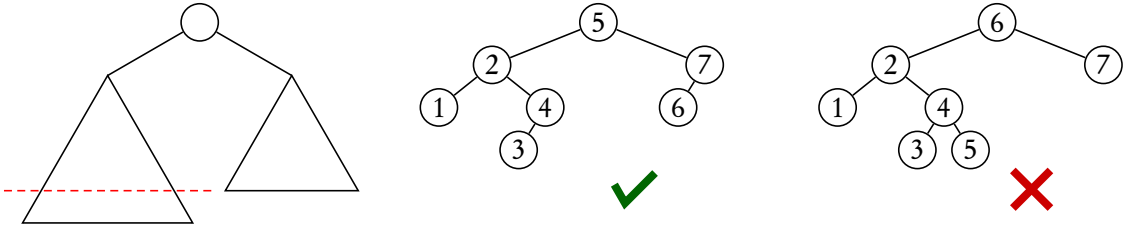
1. **AVL Trees**
   - The Condition of Balance
   - Fibonacci Trees
   - The Golden Ratio
2. **Recovering Balance**
   - Insertion
   - Tree Rotation
   - Deletion

Perfectly Balanced Trees

- A perfectly balanced tree requires that for every node, the *sizes* of its left and right subtrees differ by at most 1.
- The balance condition is too rigid. It's hard to maintain a perfectly balanced search tree.
- It is the *height* (*depth*) that actually affects the search time.
- We may relax the balance condition a little bit.

AVL Trees

An AVL tree (Adelson-Velskii and Landis, 1962) is a binary search tree, with that for every node, the heights of the left and right subtrees can differ by at most 1.



It seems to be quite balanced from its definition. It indeed is. If we try to insert new nodes to the left whenever we can (maintaining the balance condition), we may get the worst case of AVL tree. What is it?

Fibonacci Trees

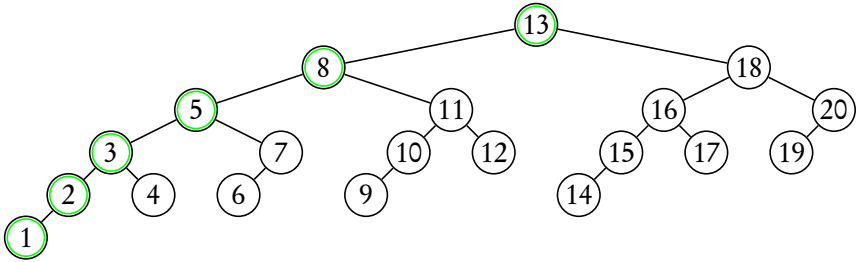
- If for every node in a tree, the height of its left subtree is always 1 more than the height of its right subtree, then it is a Fibonacci tree.
- A Fibonacci tree is
  - a leaf with height 0, or
  - a node with height 1 that has only the left subtree as a leaf, or
  - if t_1 and t_2 are Fibonacci trees with height $h-1$ and $h-2$ respectively, then the tree with t_1 as left subtree and t_2 as right subtree is a Fibonacci tree with height h.
- A Fibonacci search tree is an AVL tree. It is the worst case of AVL trees.

Sizes of Fibonacci Trees

Let the size of a Fibonacci tree of height h be n_h, we have $n_h = n_{h-1} + n_{h-2} + 1$. We can prove that

$$n_h = F_{h+3} - 1,$$

where $F_0 = 0, F_1 = 1$ and $F_i = F_{i-1} + F_{i-2}$ for $i \geqslant 2$.

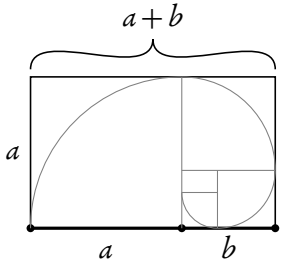


We know that the left subtree has roughly 61.8% of the nodes. It's still very balanced.

Fibonacci Numbers and the Golden Ratio

The golden ratio φ is the ratio of a larger number a to a smaller number b such that

$$\varphi = \frac{a}{b} = \frac{a+b}{a} = 1 + \frac{1}{\varphi} \implies \varphi^2 - \varphi - 1 = 0 \implies \varphi = \frac{1+\sqrt{5}}{2} = 1.6180339887\ldots$$



When $n \to +\infty$, the ratio between two consecutive Fibonacci numbers $\frac{F_n}{F_{n-1}}$ has a limit, and

$$\lim_{n\to\infty} \frac{F_n}{F_{n-1}} = \lim_{n\to\infty} \frac{F_{n+1}}{F_n} = \lim_{n\to\infty} \frac{F_n + F_{n-1}}{F_n} = \varphi.$$

The Height-Size Relation of AVL Trees

- We expect the height h of an AVL tree is logarithmic to the size n, that is, $h \in \mathcal{O}(\log_2 n)$.
- We try to prove, when $h \geqslant 0$, $\alpha \log_2 n \geqslant h$ for some constant factor $\alpha \geqslant 1$, that is $n \geqslant 2^{\frac{h}{\alpha}}$.
- We induct on h. The induction step shows,

$$n = n_{s+} + n_{s-} + 1 \geqslant 2^{\frac{h-1}{\alpha}} + 2^{\frac{h-2}{\alpha}} = \left(1 + 2^{-\frac{1}{\alpha}}\right)\left(2^{-\frac{1}{\alpha}}\right) 2^{\frac{h}{\alpha}}.$$

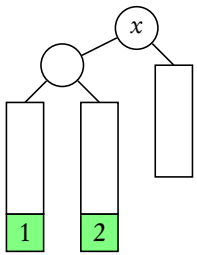
- We need $\left(1 + 2^{-\frac{1}{\alpha}}\right)\left(2^{-\frac{1}{\alpha}}\right) = 1$, that is,

$$1 + \frac{1}{2^{\frac{1}{\alpha}}} = 2^{\frac{1}{\alpha}}.$$

- Therefore, $2^{\frac{1}{\alpha}} = \varphi$, the golden ratio, thus, $\alpha = \log_\varphi 2 \approx 1.44042$.
- The fact that $h \leqslant \log_\varphi 2 \cdot \log_2 n = \log_\varphi n$ implies the Fibonacci tree is the worst case AVL tree.

Rebalancing after Insertion

- After an insertion, only nodes that are on the path from the insertion point to the root might have their balance changed. We follow this path and update the balance information.
- It's possible that a node's new balance violates the AVL condition. We need to rebalance the node.
- If a node x is to be rebalanced, that is, the heights of its subtrees differ by 2, then the insertion must have been happened to one of its children's subtrees:
  1. left subtree of x's left child (left-left), or
  2. right subtree of x's left child (left-right), or
  3. left subtree of x's right child (right-left), or
  4. right subtree of x's right child (right-right).

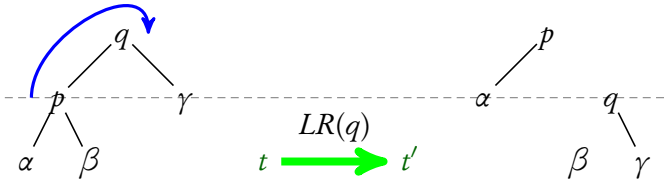


Tree Rotation

- Let p be the left child of q, α and β the left and right subtrees of p, γ the right subtree of q.
- A *left-to-right rotation* on (sub)tree t rooted at node q is that
  - let q be the right child of p;
  - let β be the left subtree of q;
  - let p be the new root of the resulted tree t'.
- After the rotation, α is shallowed and γ is deepened (by one level), and the depth of β is unchanged.
- It's symmetric for right-to-left rotations.
- *Tree rotations do not change the in-order traversal sequence of a binary tree.*

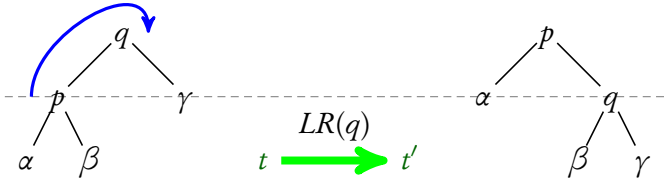
Tree Rotation

- Let p be the left child of q, α and β the left and right subtrees of p, γ the right subtree of q.
- A *left-to-right rotation* on (sub)tree t rooted at node q is that
  - let q be the right child of p;
  - let β be the left subtree of q;
  - let p be the new root of the resulted tree t'.
- After the rotation, α is shallowed and γ is deepened (by one level), and the depth of β is unchanged.
- It's symmetric for right-to-left rotations.
- *Tree rotations do not change the in-order traversal sequence of a binary tree.*



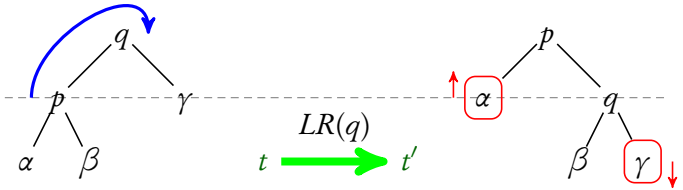
Tree Rotation

- Let p be the left child of q, α and β the left and right subtrees of p, γ the right subtree of q.
- A *left-to-right rotation* on (sub)tree t rooted at node q is that
  - let q be the right child of p;
  - let β be the left subtree of q;
  - let p be the new root of the resulted tree t'.
- After the rotation, α is shallowed and γ is deepened (by one level), and the depth of β is unchanged.
- It's symmetric for right-to-left rotations.
- *Tree rotations do not change the in-order traversal sequence of a binary tree.*

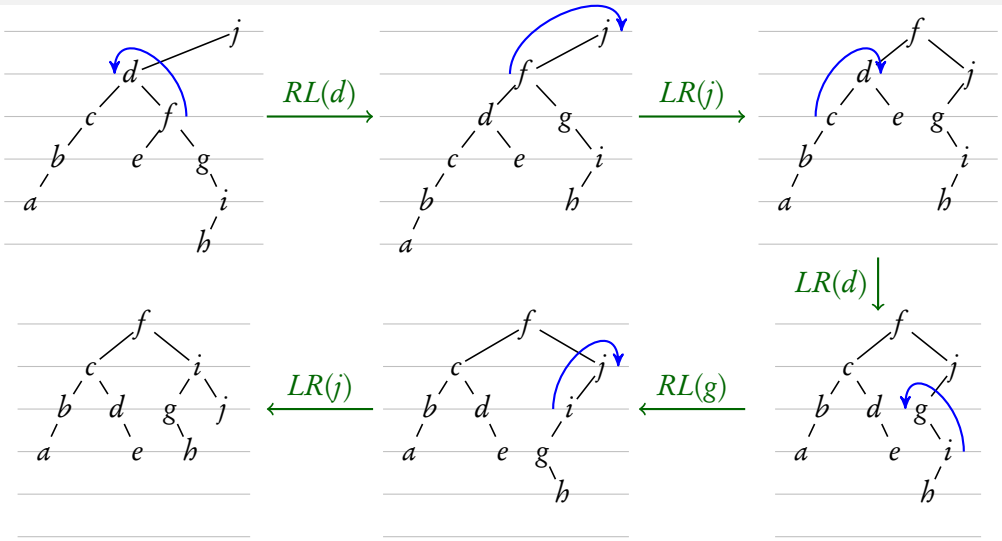


Tree Rotation

- Let p be the left child of q, α and β the left and right subtrees of p, γ the right subtree of q.
- A *left-to-right rotation* on (sub)tree t rooted at node q is that
  - let q be the right child of p;
  - let β be the left subtree of q;
  - let p be the new root of the resulted tree t'.
- After the rotation, α is shallowed and γ is deepened (by one level), and the depth of β is unchanged.
- It's symmetric for right-to-left rotations.
- *Tree rotations do not change the in-order traversal sequence of a binary tree.*

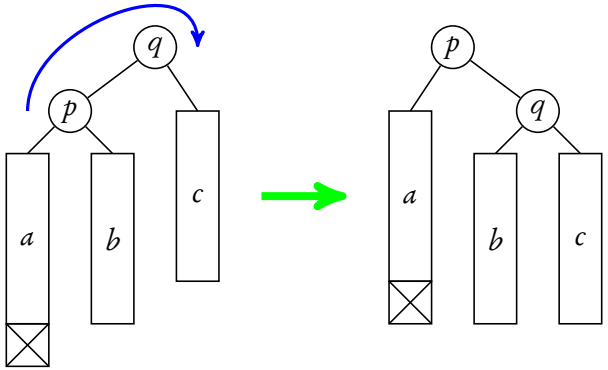


An Example of Tree Rotation



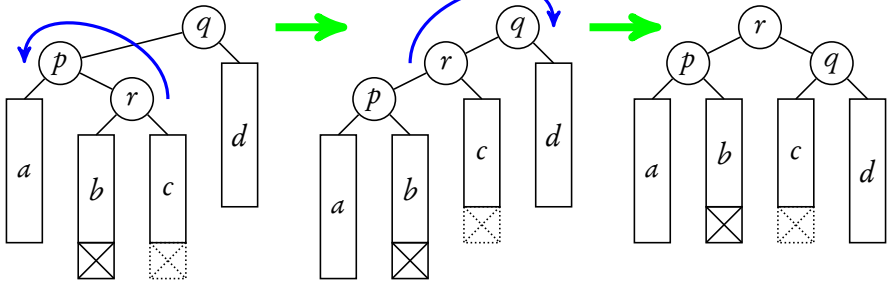
Fixing by Single Rotation

- In the cases of (1) and (4), a single rotation will fix the balance.
- These are the left-left and right-right cases.



Fixing by Double Rotation

- Let p be the left child of q, r the right child of p. A *left-to-right double rotation* on q is that
  - perform a right-to-left rotation on p;
  - perform a left-to-right rotation on q.
- There's a symmetric case from right to left.
- Case (2) and case (3) are left-right and right-left cases. A double rotation will fix the balance.



The AVL Tree Node

We add an attribute *height* to record the height of a node.

```
1  class Node:
2      def __init__(self, key, value):
3          self.key, self.value = key, value
4          self.height = 0
5          self.left = self.right = None
```

We write a *height*(*root*) function to get the height of *root*, returning −1 for if *root* is `None`.

```
def height(root):
    return -1 if root is None else root.height
```

The *update_height*(*root*) function updates the height of the *root* from its higher subtree. The function returns this *root* node for convenience.

```
def update_height(root):
    root.height = 1+max(height(root.left), height(root.right))
```

Single Rotation and Double Rotation — Code

Single rotation:

```
1  def rotate_lr(q):
2      p = q.left
3      q.left = p.right
4      p.right = q
5      update_height(q)
6      update_height(p)
7      return p
```

Fixing height by single or double rotation:

```
1  def fix_height_lr(q):
2      if height(q.left.left) < height(q.left.right):
3          q.left = rotate_rl(q.left)
4      return rotate_lr(q)
```

Fixing Heights after Making Change to Subtrees

- When we insert a node to one of the subtrees, we know which subtree is getting larger.
- When we delete a node from one of the subtrees, we know which subtree is getting smaller, and the other one is getting, relatively, larger.
- If the larger side is too high, we fix the height by tree rotations.

```
1 def left_larger(root):
2     if height(root.left)-height(root.right) == 2:
3         return fix_height_lr(root)
4     else:
5         update_height(root)
6         return root
```

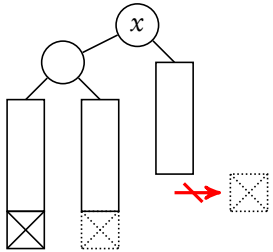
- Thus, to insert a node to the left with the height fixed, we write

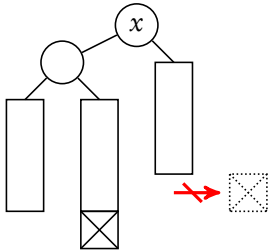
```
root.left = insert(root.left, key, value)
return left_larger(root)
```

How about Deletion?

Deletion in AVL trees is somewhat more complicated than insertion. The cases are similar, for a node x to be rebalanced:

1. Right tree is shortened, right tree of x's left child is not deeper.
2. Right tree is shortened, right tree of x's left child is deeper.
3. Left tree is shortened, left tree of x's right child is deeper.
4. Left tree is shortened, left tree of x's right child is not deeper.





More about Deletion

- Rebalancing can only shorten a tree, therefore, there are very limited rotations at insertion time.
- But at deletion time, this may cause a chain reaction, forcing all the nodes along the deletion path to be rebalanced in the worst case. Imagine deleting the right most node from a Fibonacci tree.
- There is another approach, called *lazy deletion*. It only marks the deleted nodes rather than removing them.
- For a balanced tree, removing half of the nodes will only shorten the height by 1, so keeping the deleted nodes attached won't affect the search time very much.
- And re-inserting a deleted key can save a memory allocation overhead.