AVL Trees

Some slides from Eric Roberts, CS 106B, Stanford

7.二叉搜索树

(a) 概述

There's nothing in your head
the sorting hat can't see.
So try me on and I will tell
you where you ought to be.

- Harry Potter and the Sorcerer's Stone

邓俊辉

deng@tsinghua.edu.cn

查找

- 按照事先约定的规则，从数据集合中找出符合特定条件的对象
- 对于算法的构建而言，属于最为基本而重要的静态操作
- 很遗憾，基本的数据结构并不能高效地兼顾静态查找与动态修改

基本结构	查找	插入/删除
无序向量	O(n)	O(n)
有序向量	O(logn)	O(n)
无序列表	O(n)	O(1)
有序列表	O(n)	O(n)

- 那么，能否综合现有方法的优点？如何做到？

循关键码访问

- 数据项之间，依照各自的关键码彼此区分

  call-by-key

- 条件：关键码之间支持

  大小比较与

  相等比对

- 数据集合中的数据项

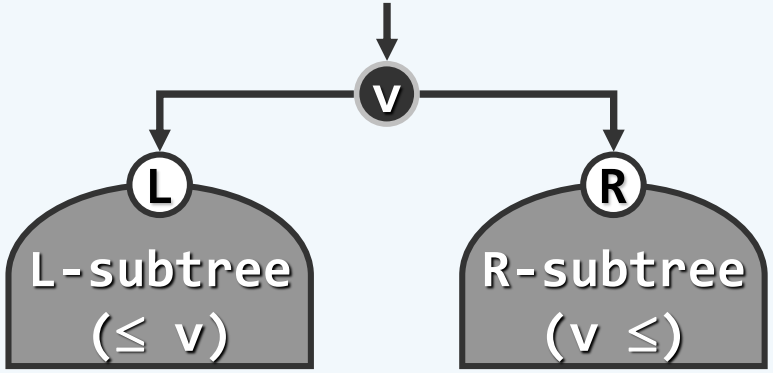
  统一地表示和实现为词条entry形式

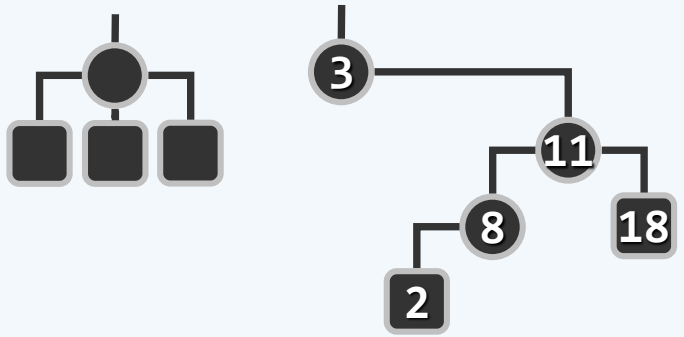
词条

```
❖ template <typename K, typename V> struct Entry { //词条模板类
     K key; V value; //关键码、数值
     Entry( K k = K(), V v = V() ) : key(k), value(v) {}; //默认构造函数
     Entry( Entry<K, V> const & e ) : key(e.key), value(e.value) {}; //克隆
   // 比较器、判等器（从此，不必严格区分词条及其对应的关键码）
     bool operator< ( Entry<K, V> const & e ) { return key < e.key; } //小于
     bool operator> ( Entry<K, V> const & e ) { return key > e.key; } //大于
     bool operator==( Entry<K, V> const & e ) { return key == e.key; } //等于
     bool operator!=( Entry<K, V> const & e ) { return key != e.key; } //不等
   };
```

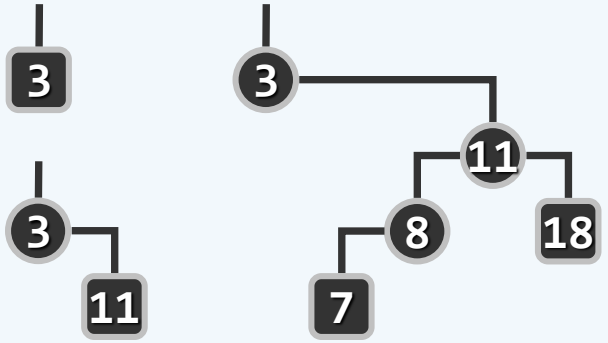
顺序性

❖ Binary Search Tree：节点 ~ 词条 ~ 关键码

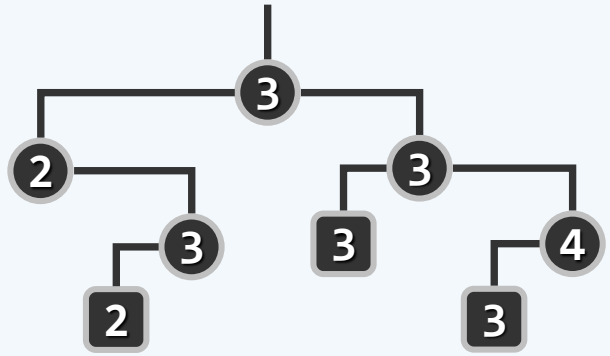




❖ 顺序性：任一节点均不小于/不大于其左/右后代



❖ 等效？：任一节点均不小于/不大于其左/右孩子



❖ 为简化起见，禁止重复词条

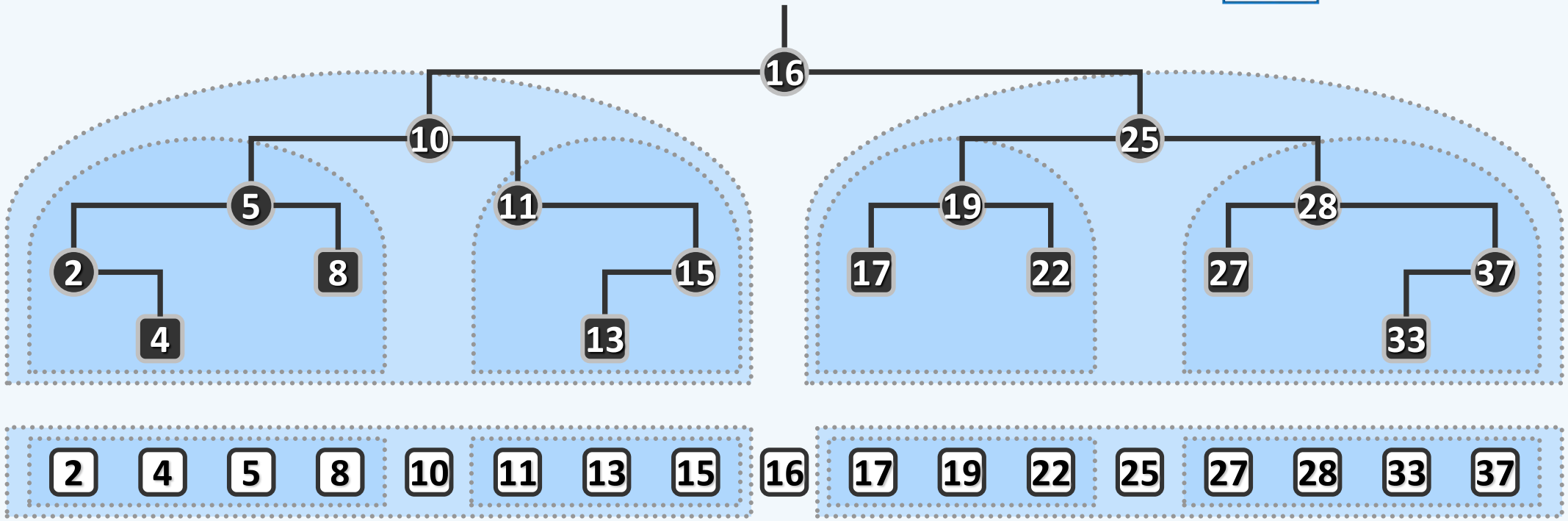
❖ 这种简化：应用中不自然，算法上无必要

习题解析：[7-10] + [7-13] + [8-3]

单调性

- 顺序性虽然只是对局部特征的刻画，但由此却可导出某种全局特征...
- 单调性：BST的中序遍历序列，必然单调非降
- 这一性质，也是BST的充要条件

//对树高做数学归纳...



BST模板类

```
❖template <typename T> class BST : public BinTree<T> { //由BinTree派生
  public: //以virtual修饰，以便派生类重写
     virtual BinNodePosi(T) & search( const T & ); //查找
     virtual BinNodePosi(T) insert( const T & ); //插入
     virtual bool remove( const T & ); //删除
  protected:
     BinNodePosi(T) _hot; //命中节点的父亲
     BinNodePosi(T) connect34( //3 + 4重构
        BinNodePosi(T), BinNodePosi(T), BinNodePosi(T),
        BinNodePosi(T), BinNodePosi(T), BinNodePosi(T), BinNodePosi(T));
     BinNodePosi(T) rotateAt( BinNodePosi(T) ); //旋转调整
  };
```

7.二叉搜索树

（b）平衡与等价

邓俊辉

deng@tsinghua.edu.cn

树高

- 由以上的实现与分析，BST主要接口search()、insert()和remove()的运行时间

  在最坏情况下，均线性正比于其高度$O(h)$

- 因此，若不能有效地控制树高，则就实际的性能而言

  较之此前的向量和列表等数据结构，BST将无法体现出明显优势

- 比如在最坏情况下，二叉搜索树可能彻底地**退化**为列表

  此时的查找效率甚至会降至$O(n)$，线性正比于树（列表）的规模

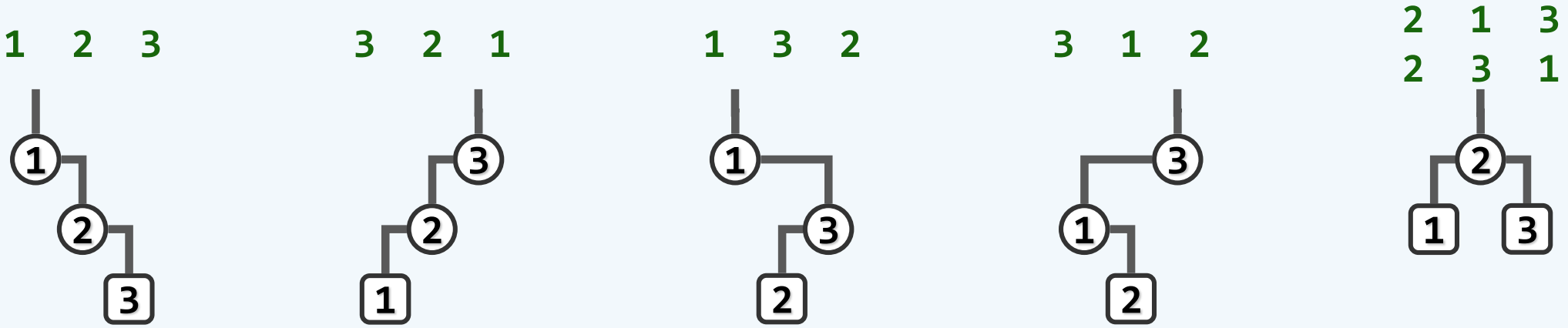
- 那么，出现此类最坏或较坏情况的概率有多大？

  或者，从平均复杂度的角度看，二叉搜索树的性能究竟如何呢？

- 以下按两种常用的随机统计**口径**，就BST的**平均性能**做一分析和对比

随机生成

- 考查n个互异词条{ e_1, e_2, ..., e_n }，对任一排列σ = (e_{i1}, e_{i2}, ..., e_{in}) ...
- 从空树开始，反复调用insert()接口将各词条**依次插入**，得到T(σ)
- 与σ相对应的T(σ)，称由σ**随机生成** (randomly generated)

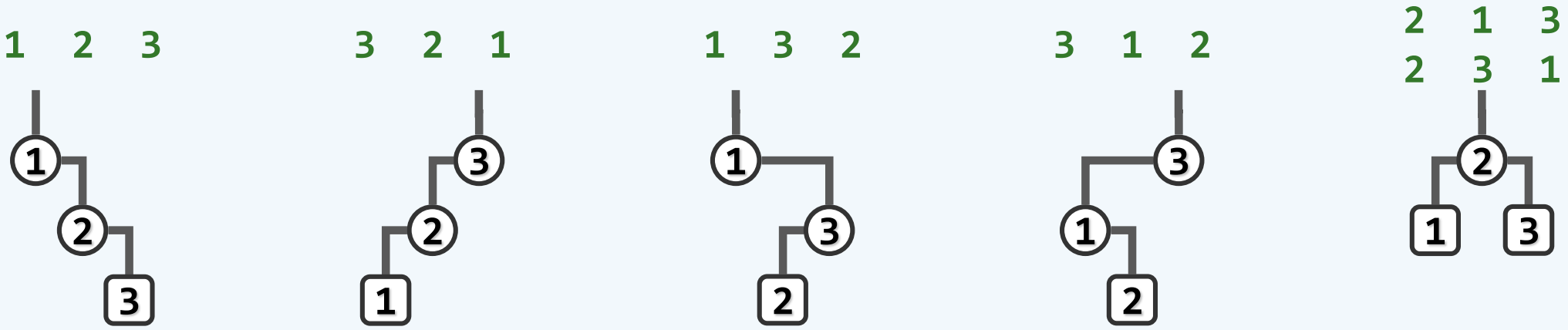


- 若假定任一排列σ作为输入的概率均等1/n!

  则由n个互异词条随机生成的二叉搜索树，平均高度为Θ(logn)

随机组成

- n个互异节点，在遵守顺序性的前提下，可随机确定拓扑联接关系
- 如此所得的BST，称由这组节点**随机组成** (randomly composed)



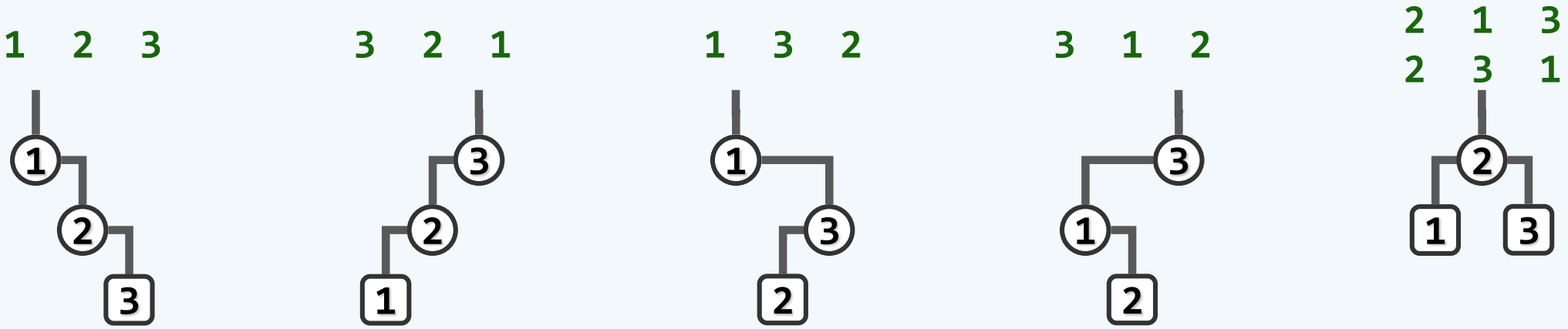
- 由n个互异节点随机组成的BST，若共计T(n)棵，则有

$$T(n) = \sum_{k=1}^{n} SP(k-1) \cdot SP(n-k) = \text{Catalan}(n) = (2n)! / (n+1)! / n!$$

- 假定所有BST等概率出现，则其平均高度为 $\Theta(\sqrt{n})$

Θ(logn) vs. Θ(sqrt(n))

- 按两种口径所估计的平均性能，差异极大——谁更可信？谁更接近于真实情况？

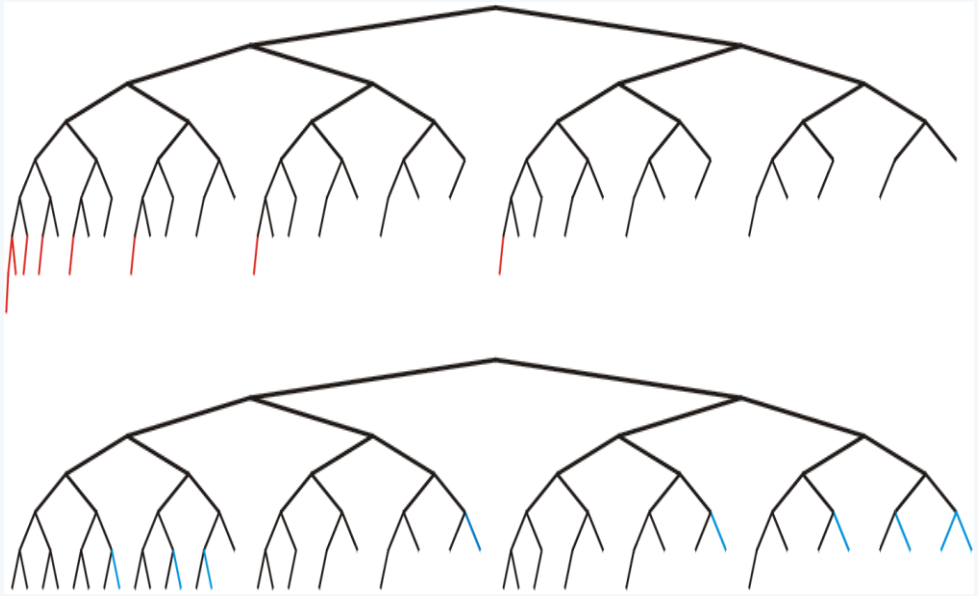


- 前一口径中，越低的BST被重复统计更多次——故嫌过于乐观
- 若删除算法固定使用succ()，则每棵BST都有越来越左倾的趋势
- 理想随机在实际中并不常见，关键码往往按单调甚至线性的次序出现

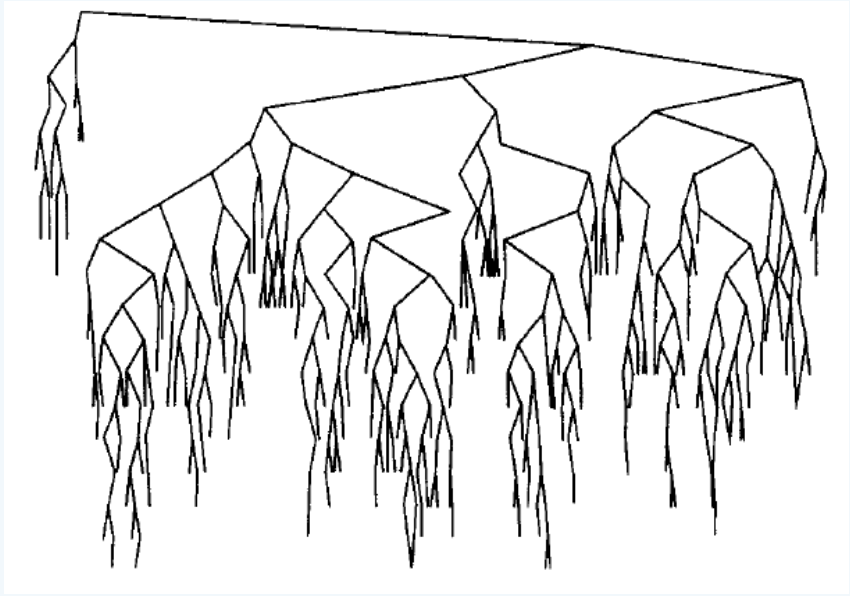
  极高的BST频繁出现，不足为怪

Height of an AVL Tree

In this example, $n = 88$



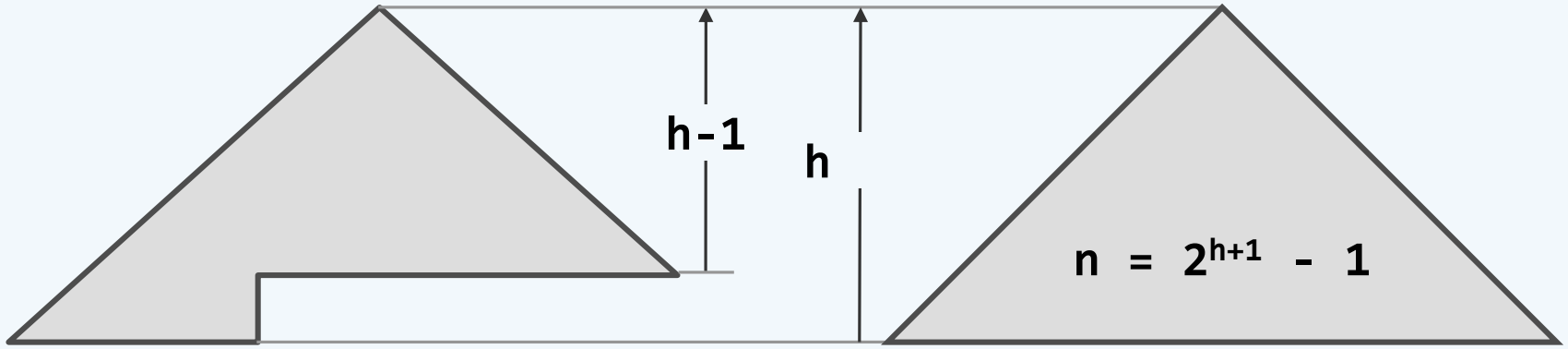
(a) AVL



(b) Random BST

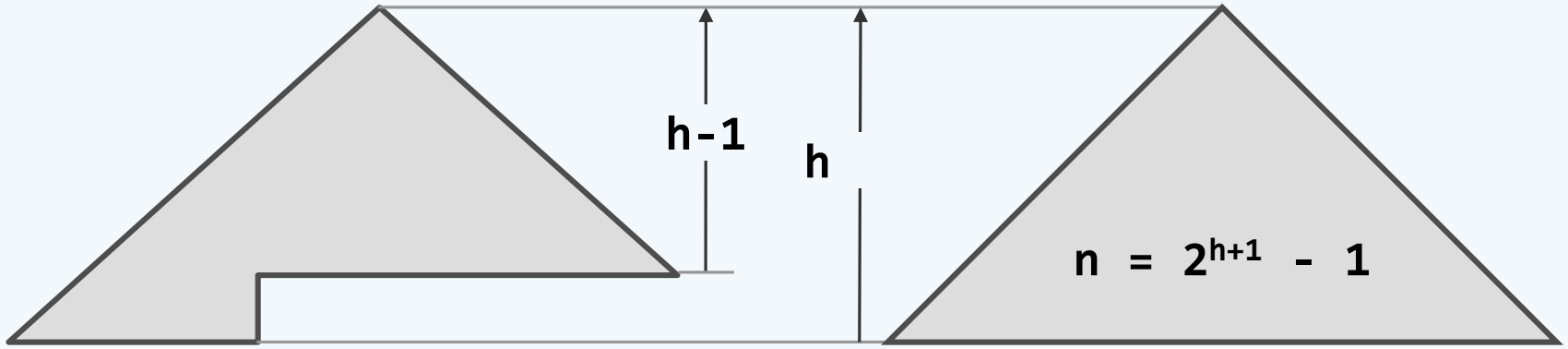
理想平衡

- 节点数目固定时，兄弟子树高度越接近（平衡），全树也将倾向于更低
- 由n个节点组成的二叉树，高度不低于$\lfloor \log_2 n \rfloor$——恰为$\lfloor \log_2 n \rfloor$时，称作理想平衡
- 大致相当于完全树甚至满树：叶节点只能出现于最底部的两层——条件过于苛刻



适度平衡

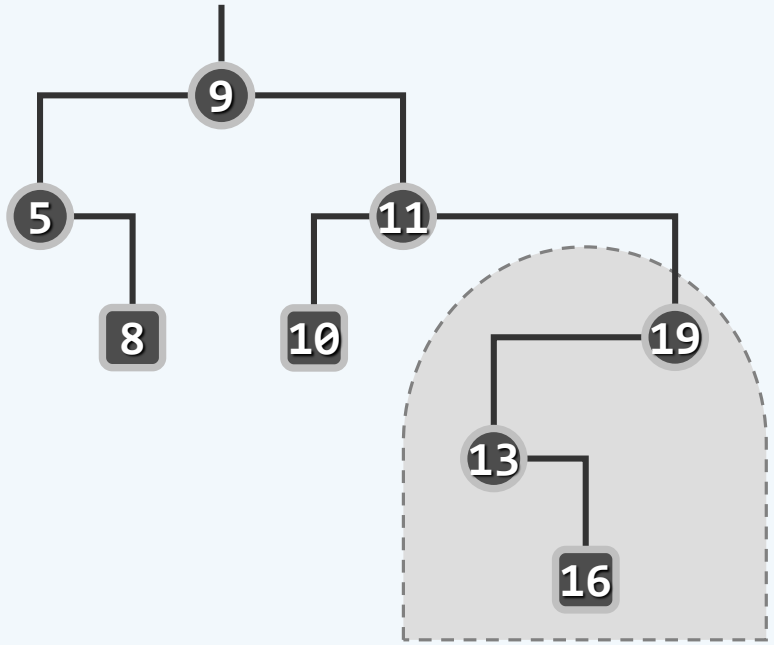
- 理想平衡出现概率极低、维护成本过高，故须适当地放松标准
- 退一步海阔天空：高度渐进地不超过$O(\log n)$，即可称作适度平衡
- 适度平衡的BST，称作平衡二叉搜索树（BBST）



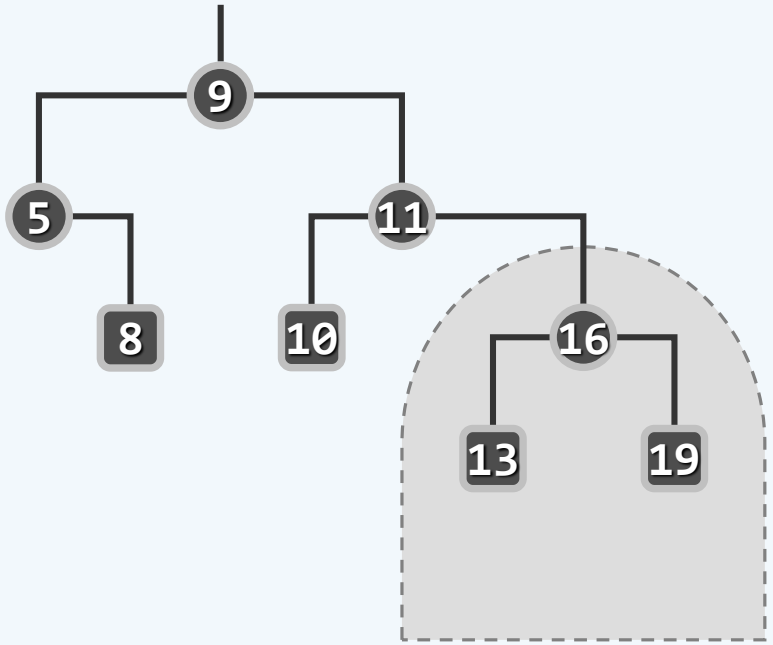
等价BST

❖ **上下可变**：联接关系不尽相同，承袭关系可能颠倒

左右不乱：中序遍历序列完全一致，全局单调非降



5 8 9 10 11 13 16 19



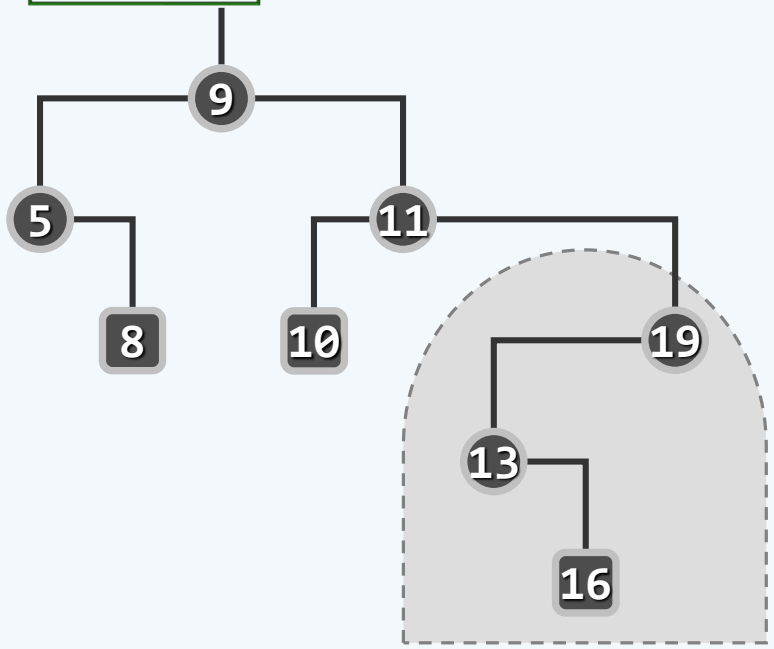
5 8 9 10 11 13 16 19

限制条件 + 局部性

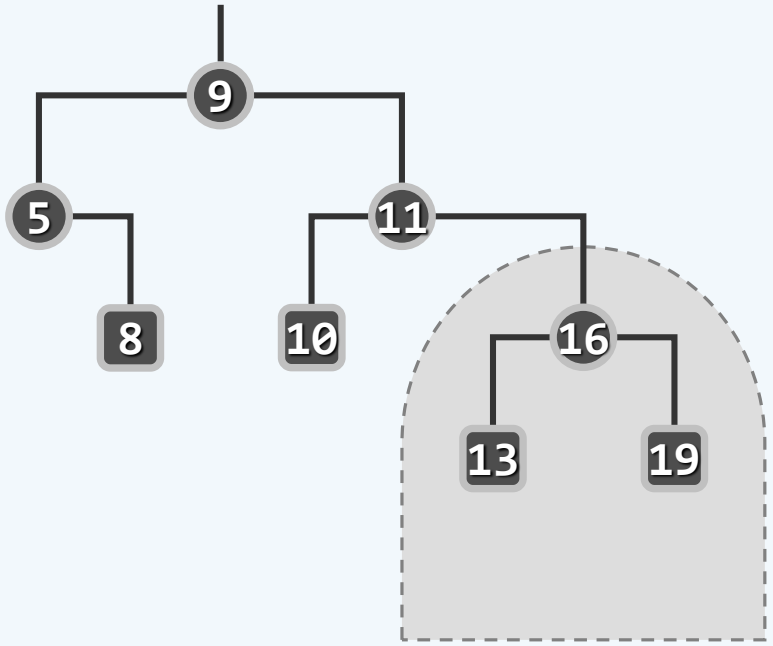
❖ 各种BBST都可视作BST的某一子集，相应地满足精心设计的限制条件

1）单次动态修改操作后，至多$O(1)$处局部不再满足限制条件

2）可在$O(\log n)$时间内，使这些局部（以至全树）重新满足



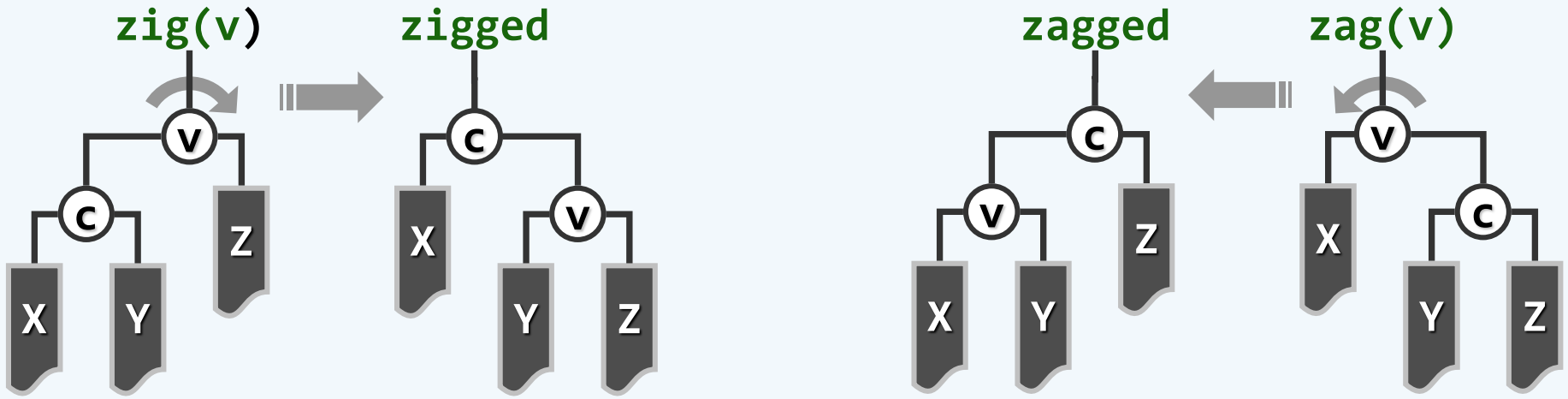
5 8 9 10 11 13 16 19



5 8 9 10 11 13 16 19

等价变换 + 旋转调整

❖ 刚刚失衡的BST，必可迅速转换为一棵等价的BBST——为此，只需O(logn)甚至O(1)次旋转



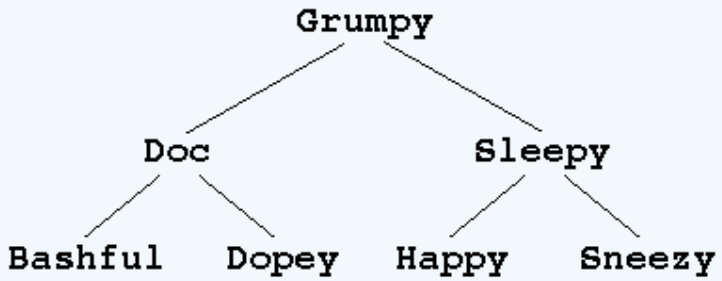
❖ zig和zag：仅涉及常数个节点，只需调整其间的联接关系；均属于局部操作、基本操作

❖ 调整之后：v/c深度加/减1，子（全）树高度的变化幅度，上下不超过1

❖ 实际上，经过不超过O(n)次旋转，等价的BST均可相互转化 习题解析[7-15]

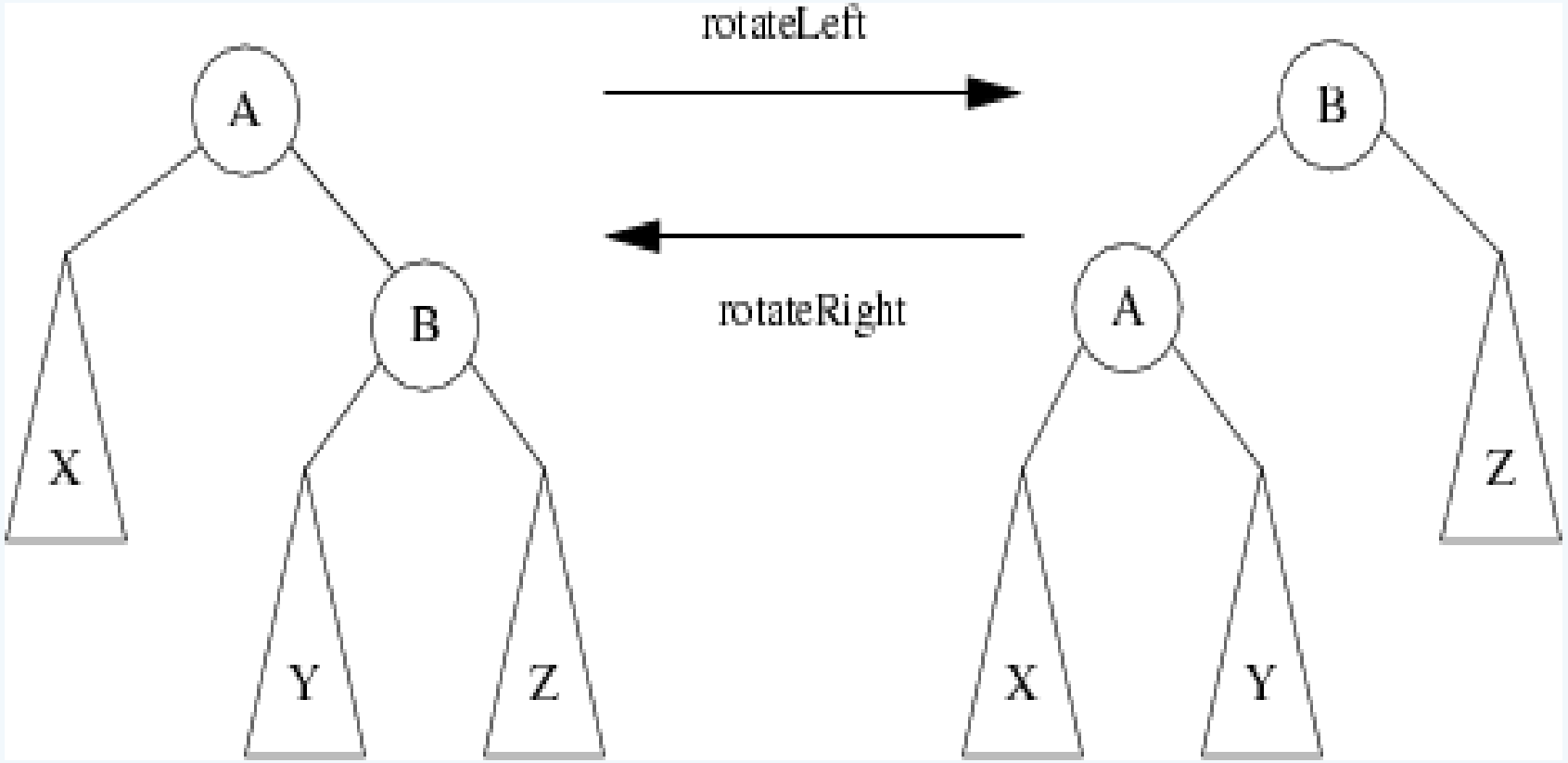
A Question of Balance

- Ideally, a binary search tree containing the names of Disney's seven dwarves would look like this:



- If, however, you happened to enter the names in alphabetical order, this tree would end up being a simple linked list in which all the left subtrees were `NULL` and the right links formed a simple chain. Algorithms on that tree would run in $O(N)$ time instead of $O(\log N)$ time.
- A binary search tree is *balanced* if the height of its left and right subtrees differ by at most one and if both of those subtrees are themselves balanced.

Rotate Left and Right



Illustrating the AVL Algorithm

H

He

Li

Be

B

C

Illustrating the AVL Algorithm

H

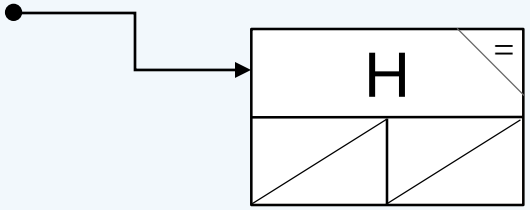
He

Li

Be

B

C



Illustrating the AVL Algorithm

H

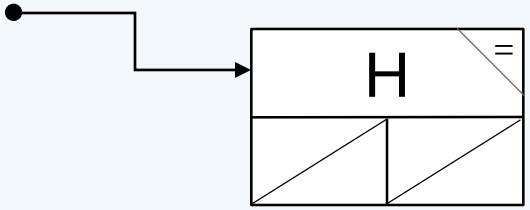
He

Li

Be

B

C



Illustrating the AVL Algorithm

H

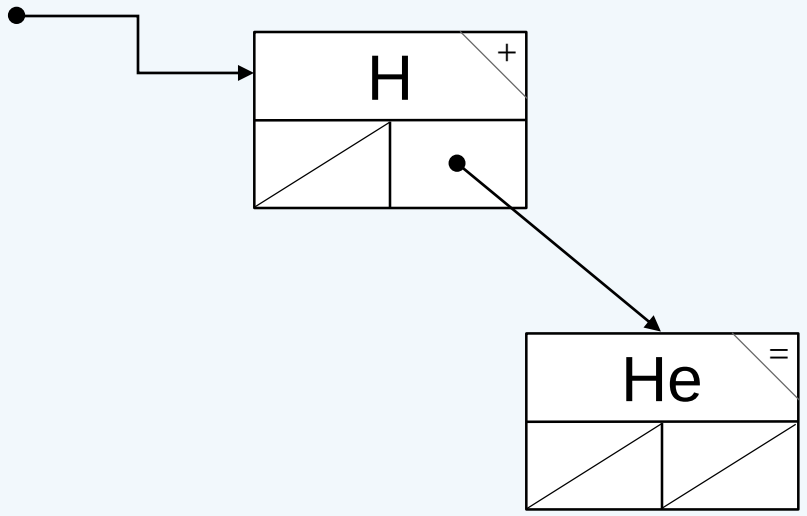
He

Li

Be

B

C



Illustrating the AVL Algorithm

H

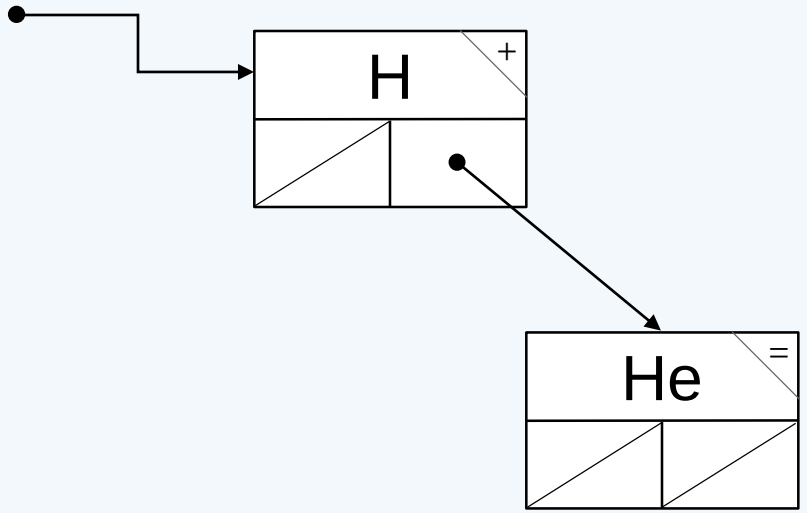
He

Li

Be

B

C



Illustrating the AVL Algorithm

H

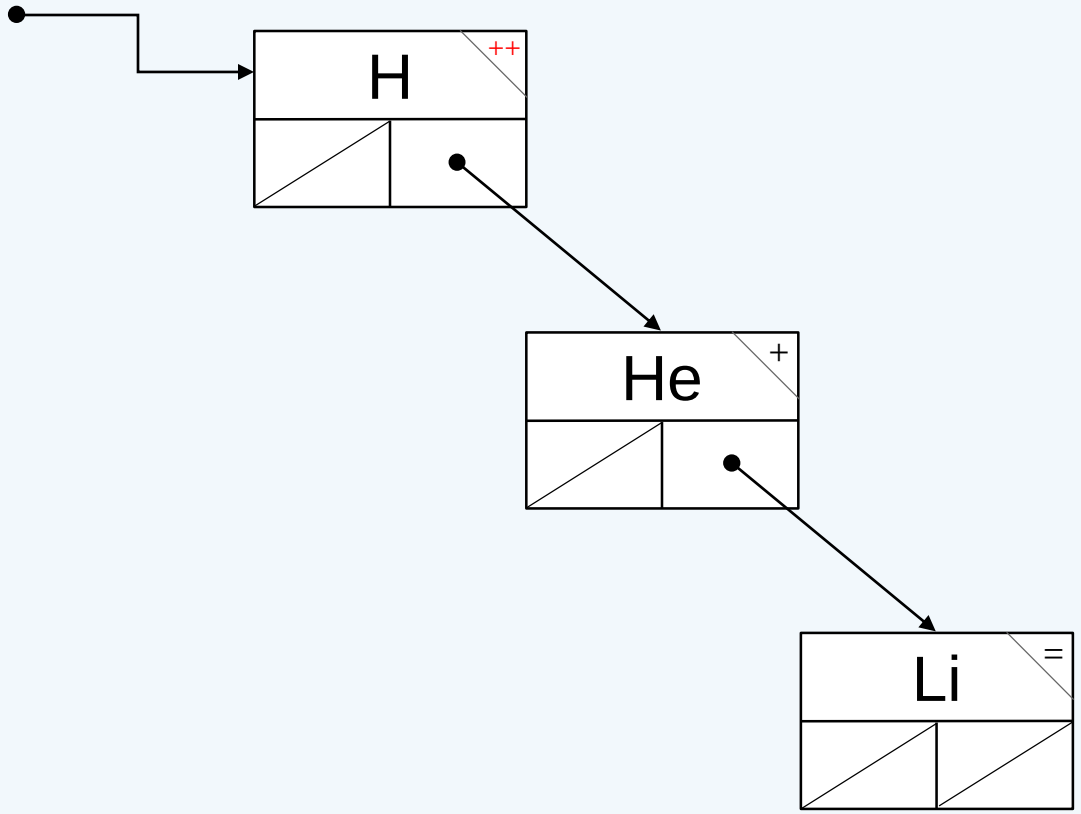
He

Li

Be

B

C



Illustrating the AVL Algorithm

H

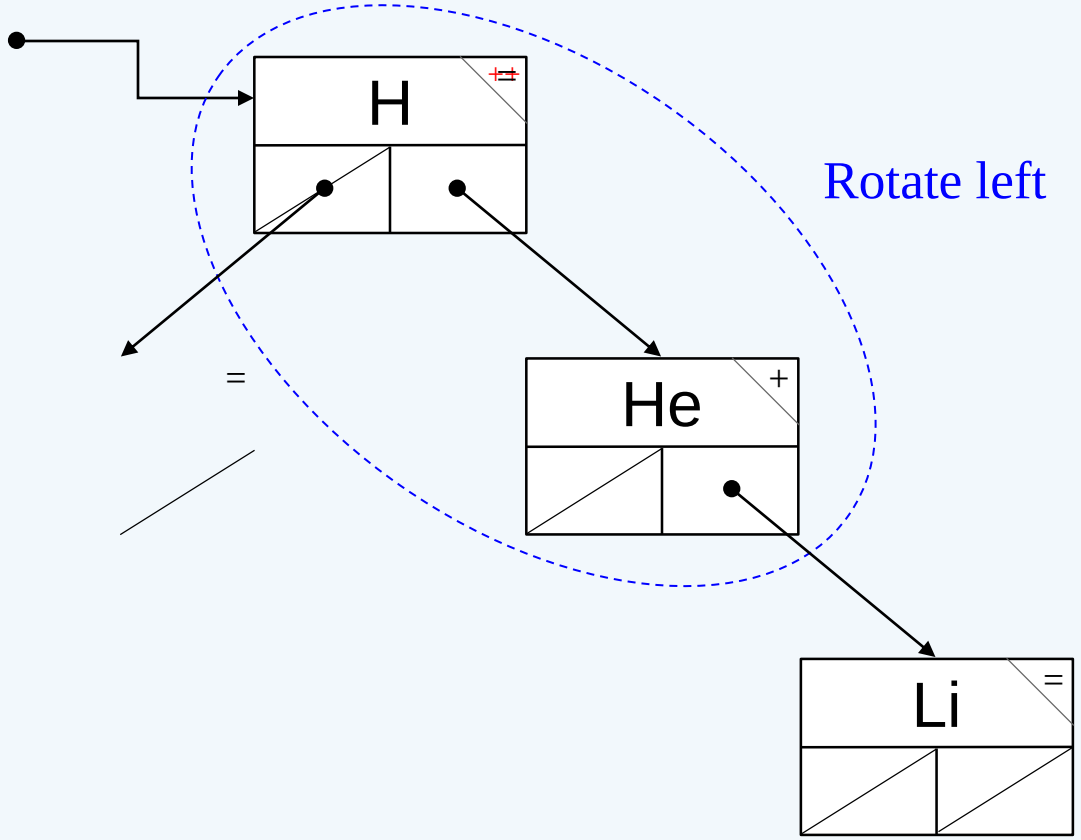
He

Li

Be

B

C



Illustrating the AVL Algorithm

H

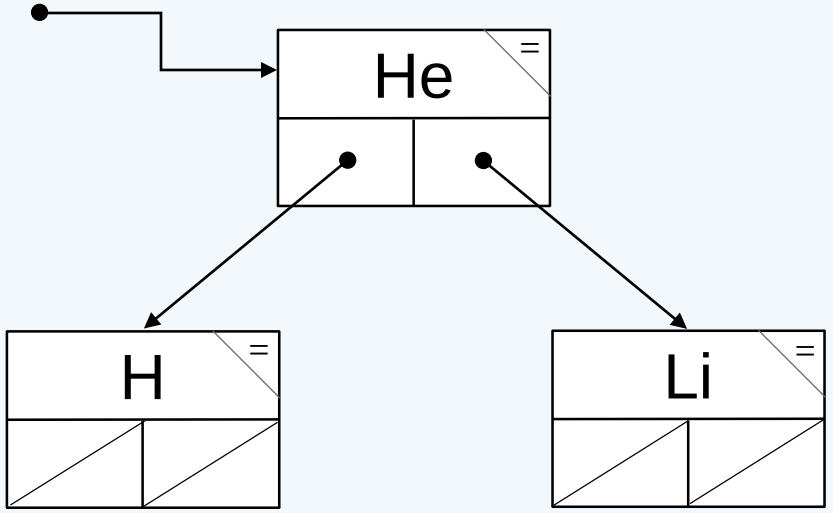
He

Li

Be

B

C



Illustrating the AVL Algorithm

H

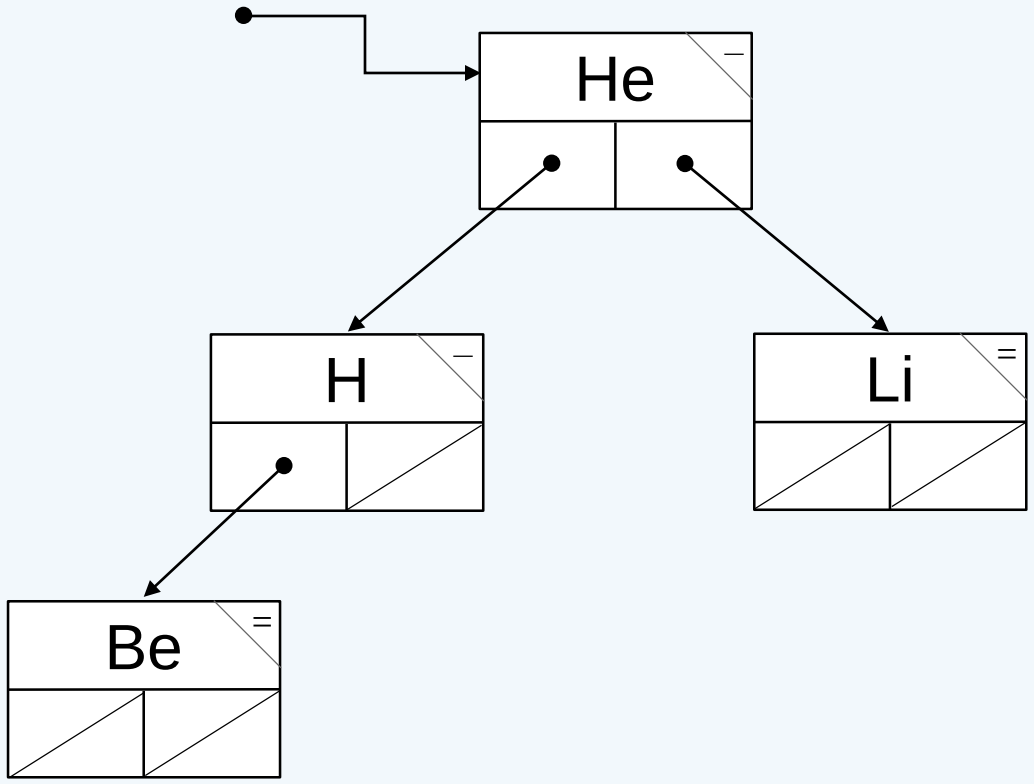
He

Li

Be

B

C



Illustrating the AVL Algorithm

H

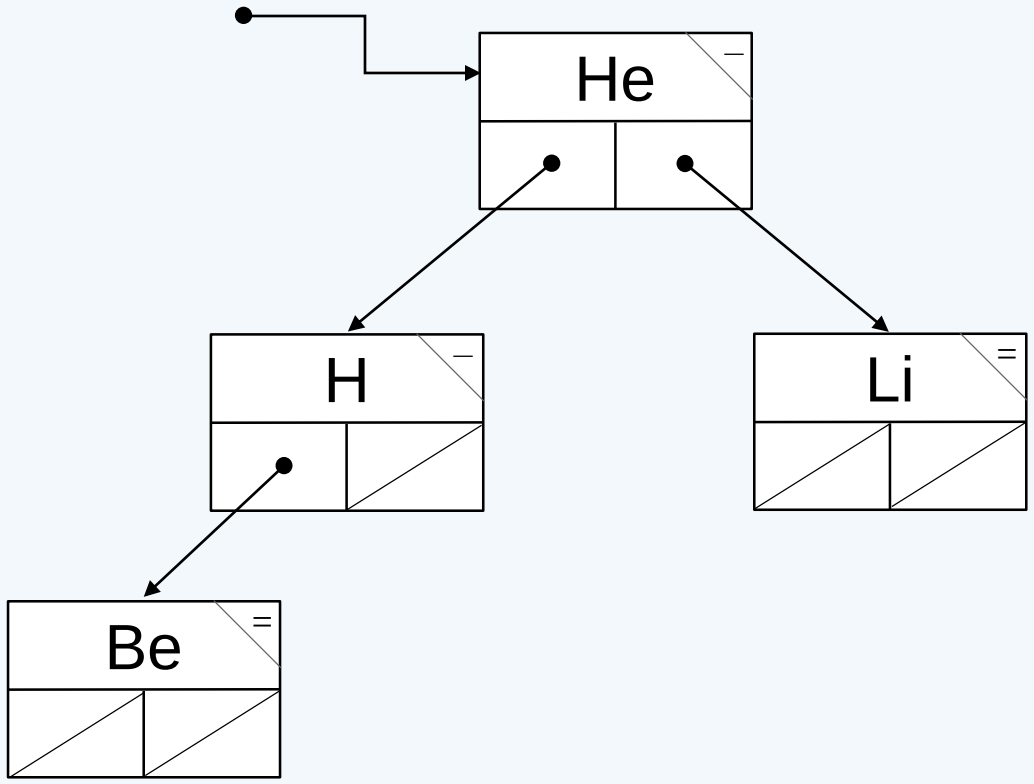
He

Li

Be

B

C



Illustrating the AVL Algorithm

H

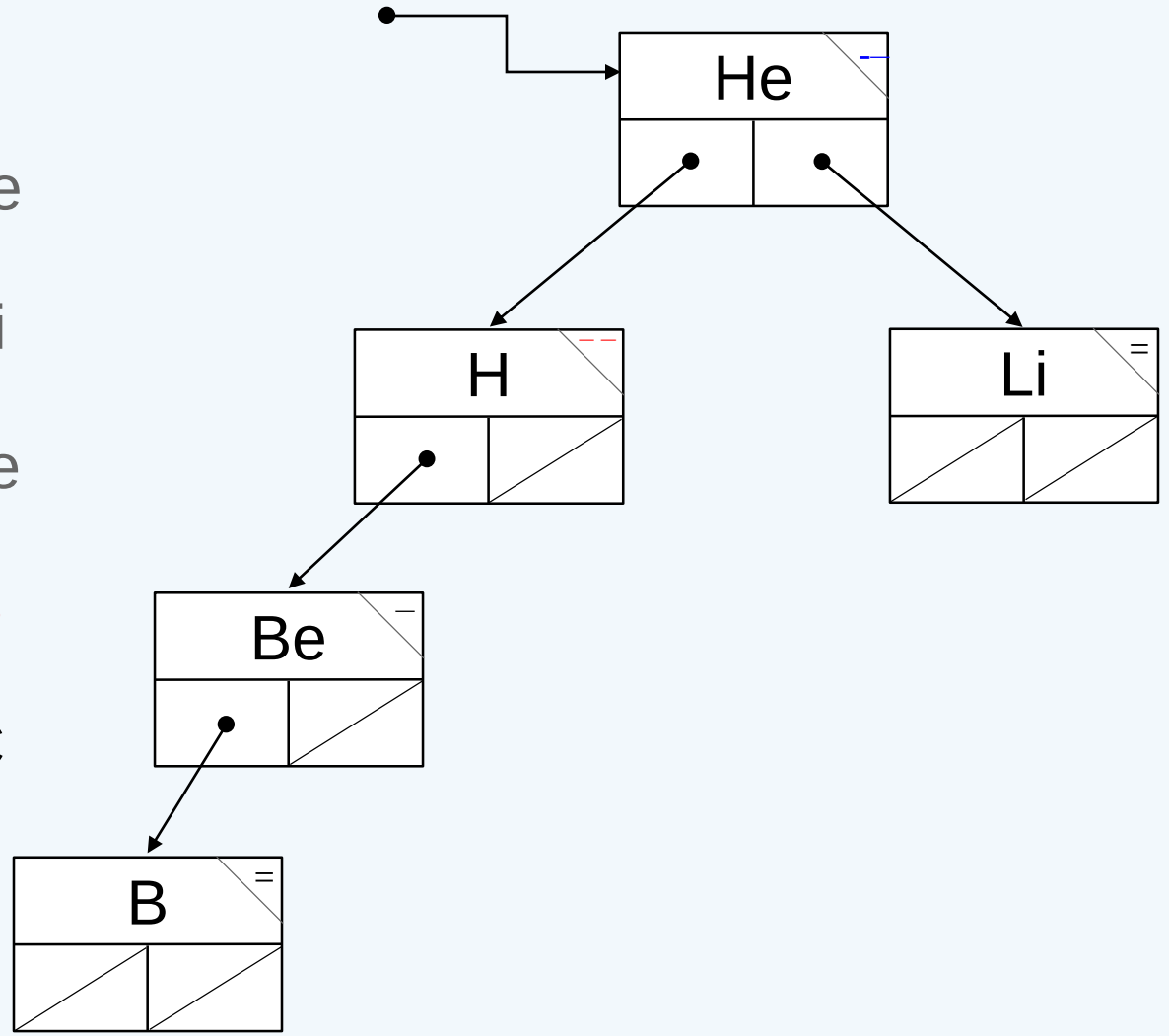
He

Li

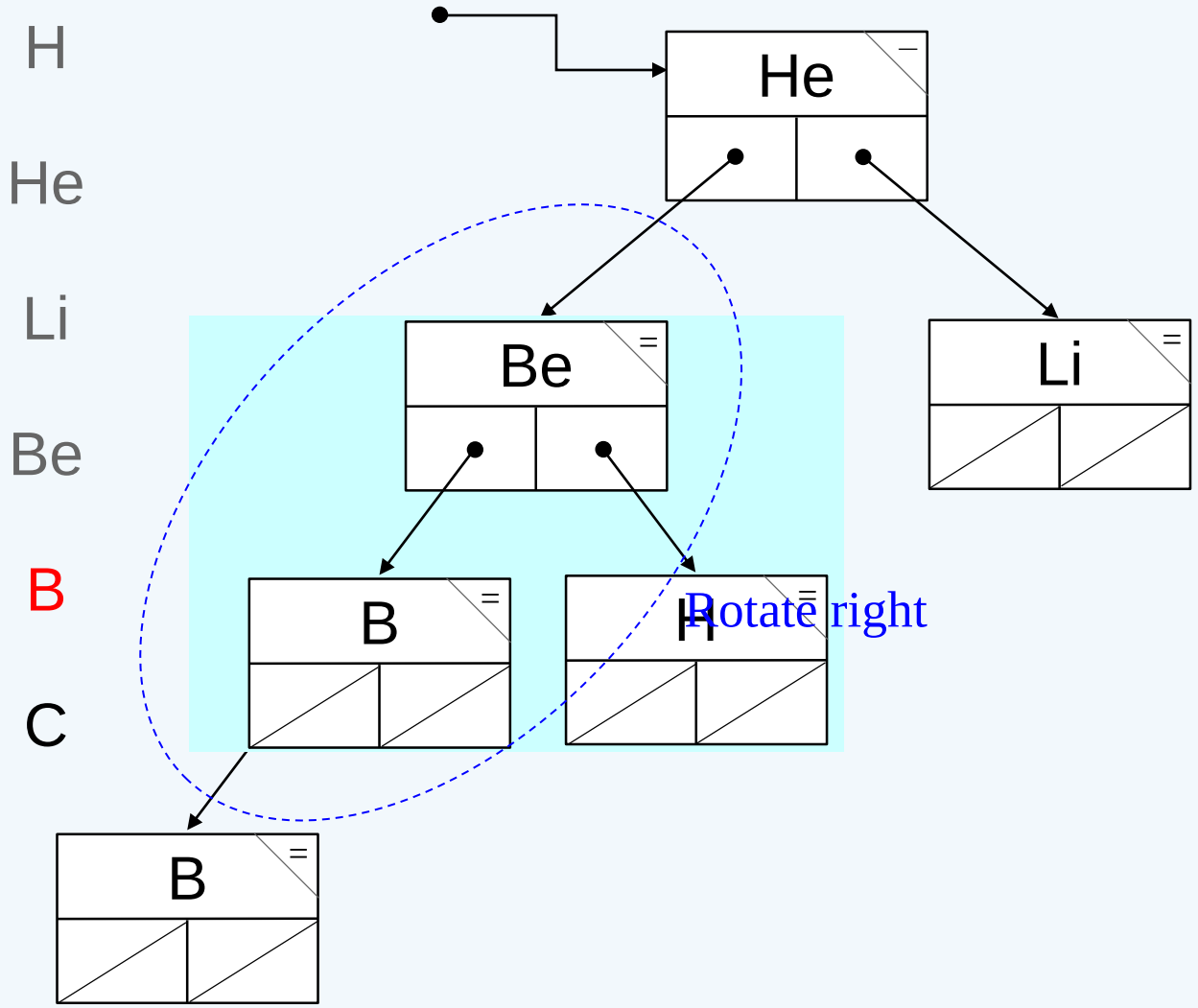
Be

B

C



Illustrating the AVL Algorithm



Illustrating the AVL Algorithm

H

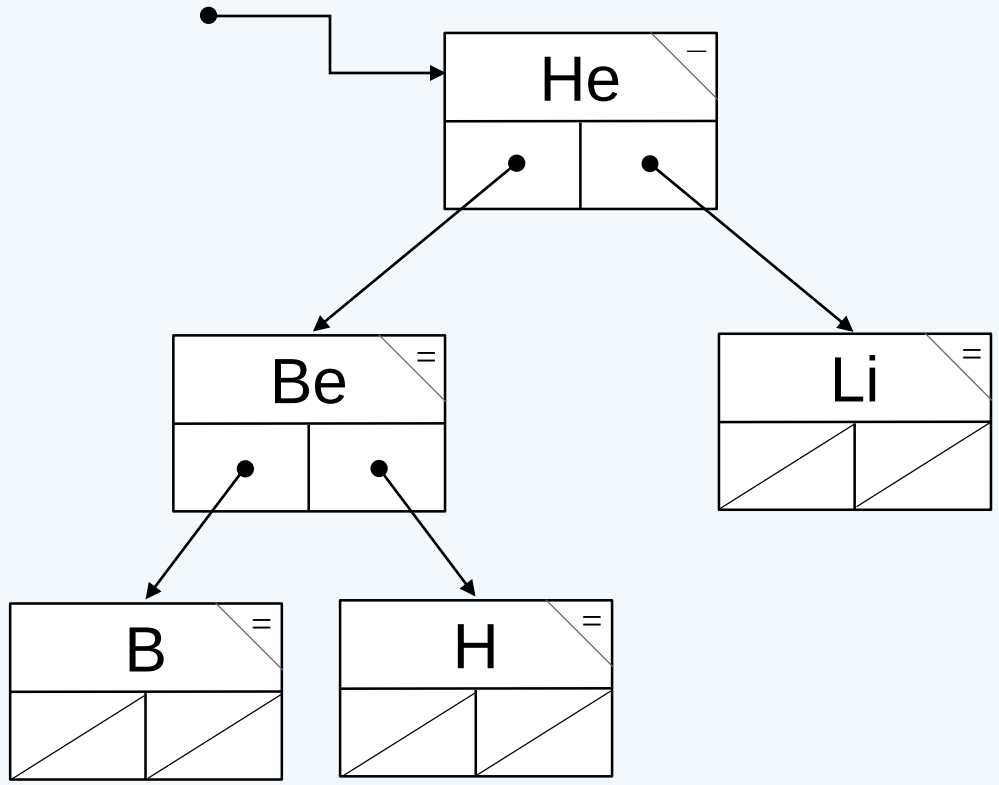
He

Li

Be

B

C



Illustrating the AVL Algorithm

H

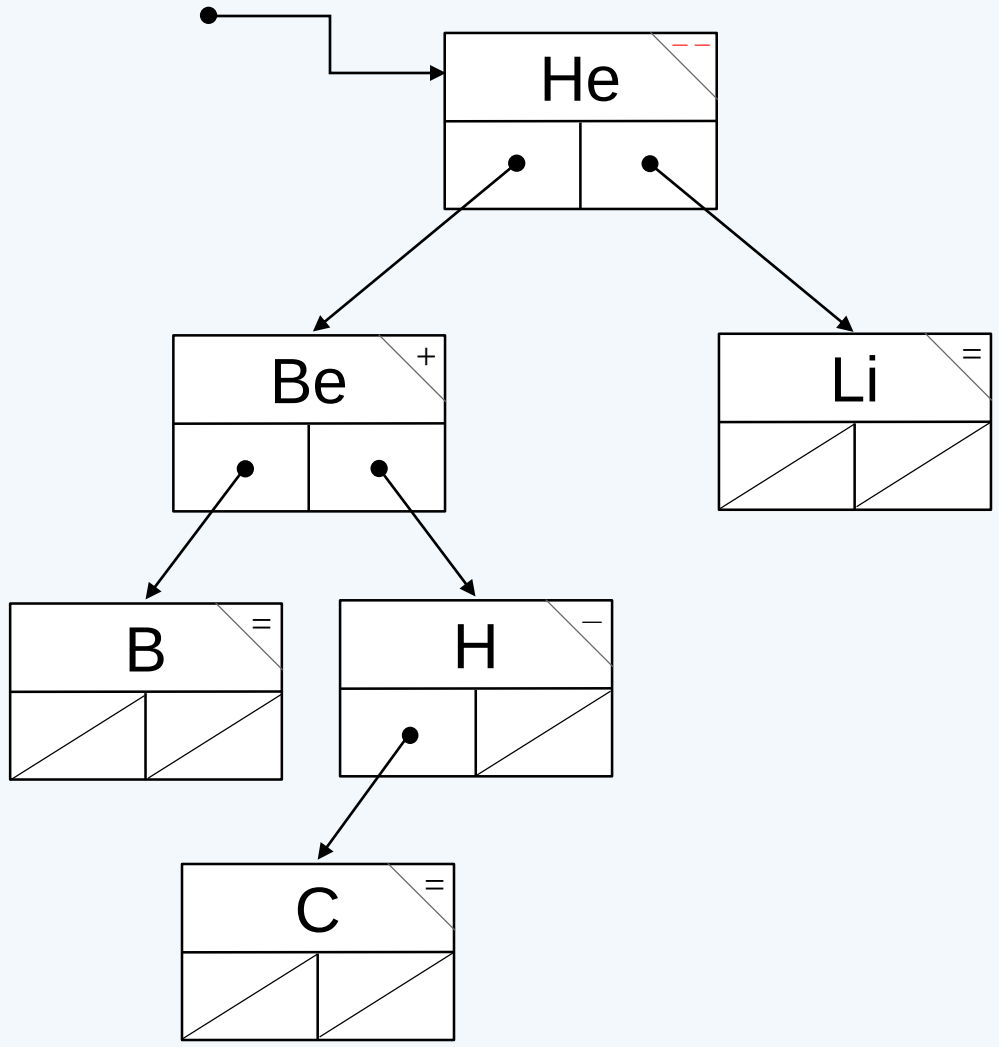
He

Li

Be

B

C



Illustrating the AVL Algorithm

H

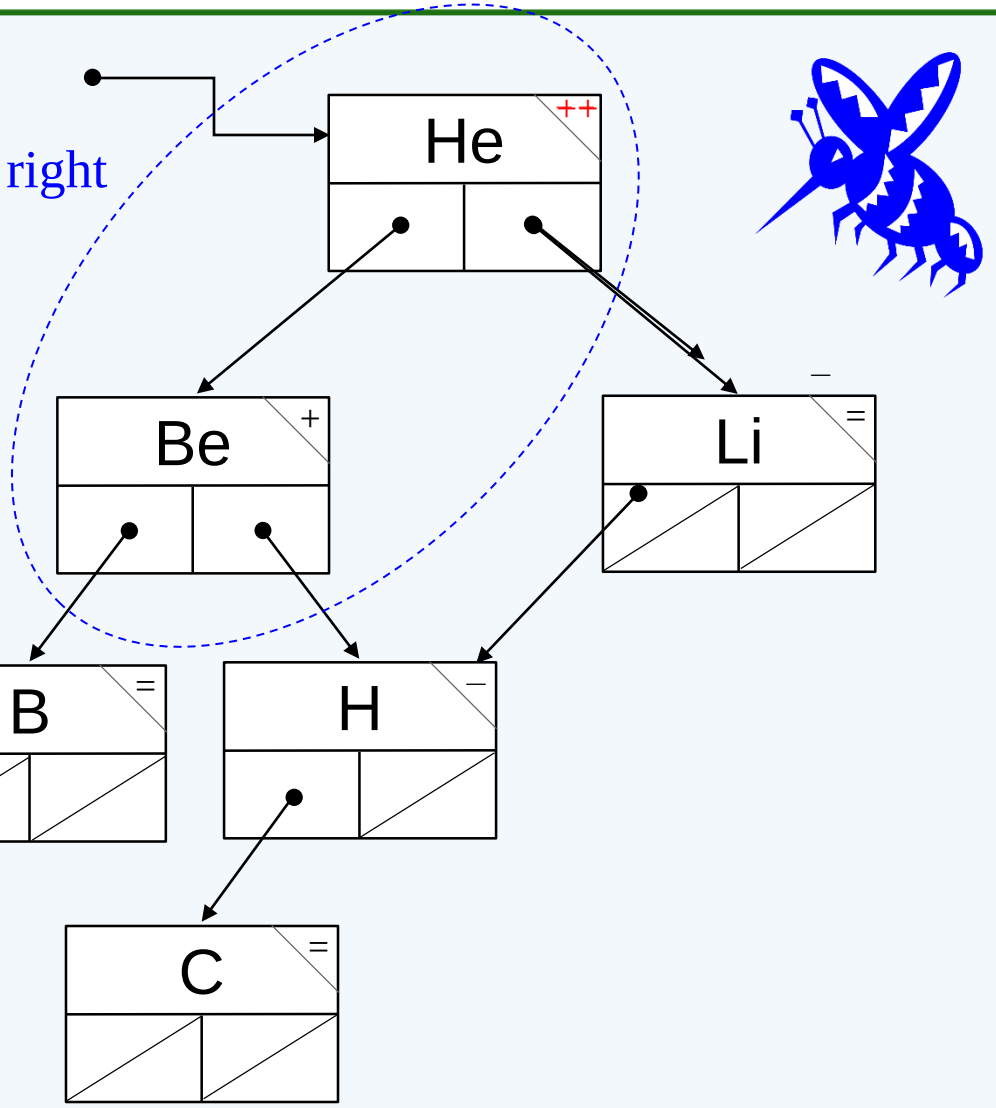
He

Li

Be

B

C



Illustrating the AVL Algorithm

H

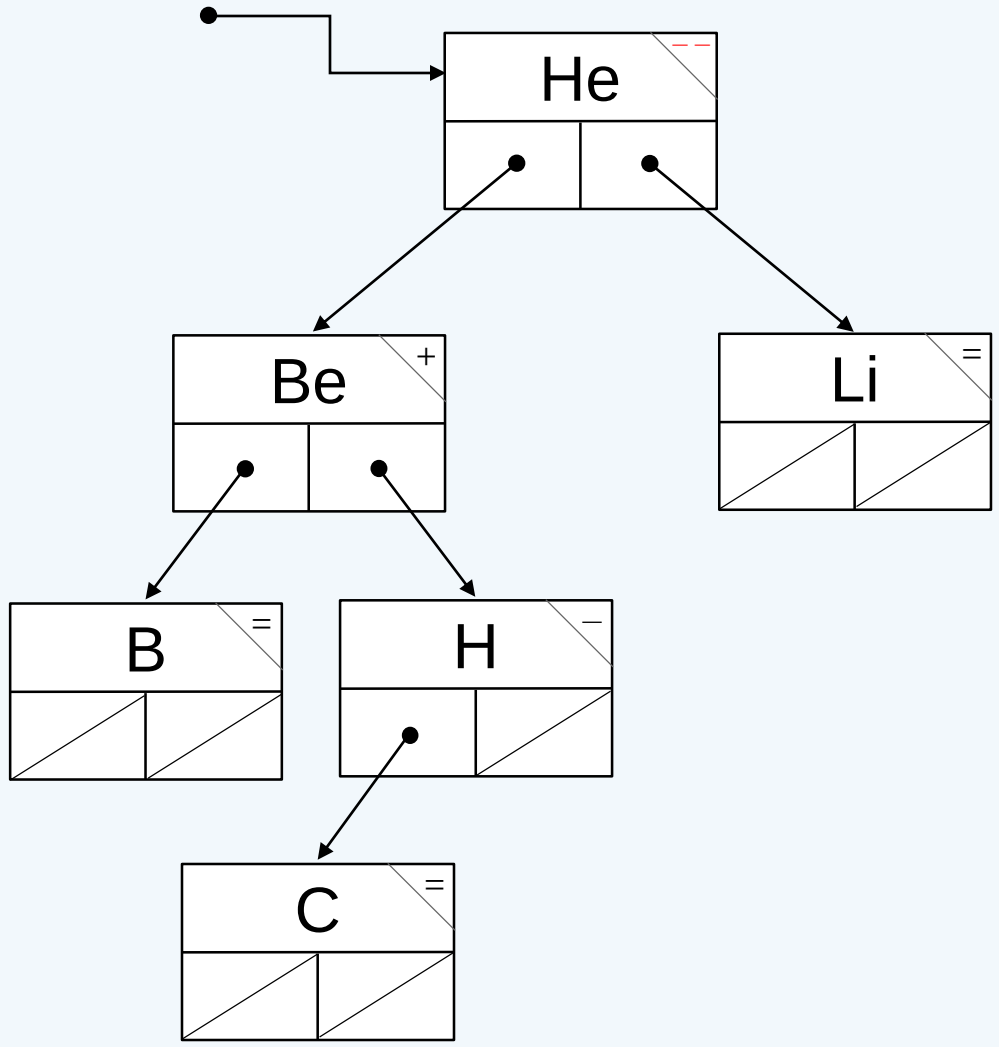
He

Li

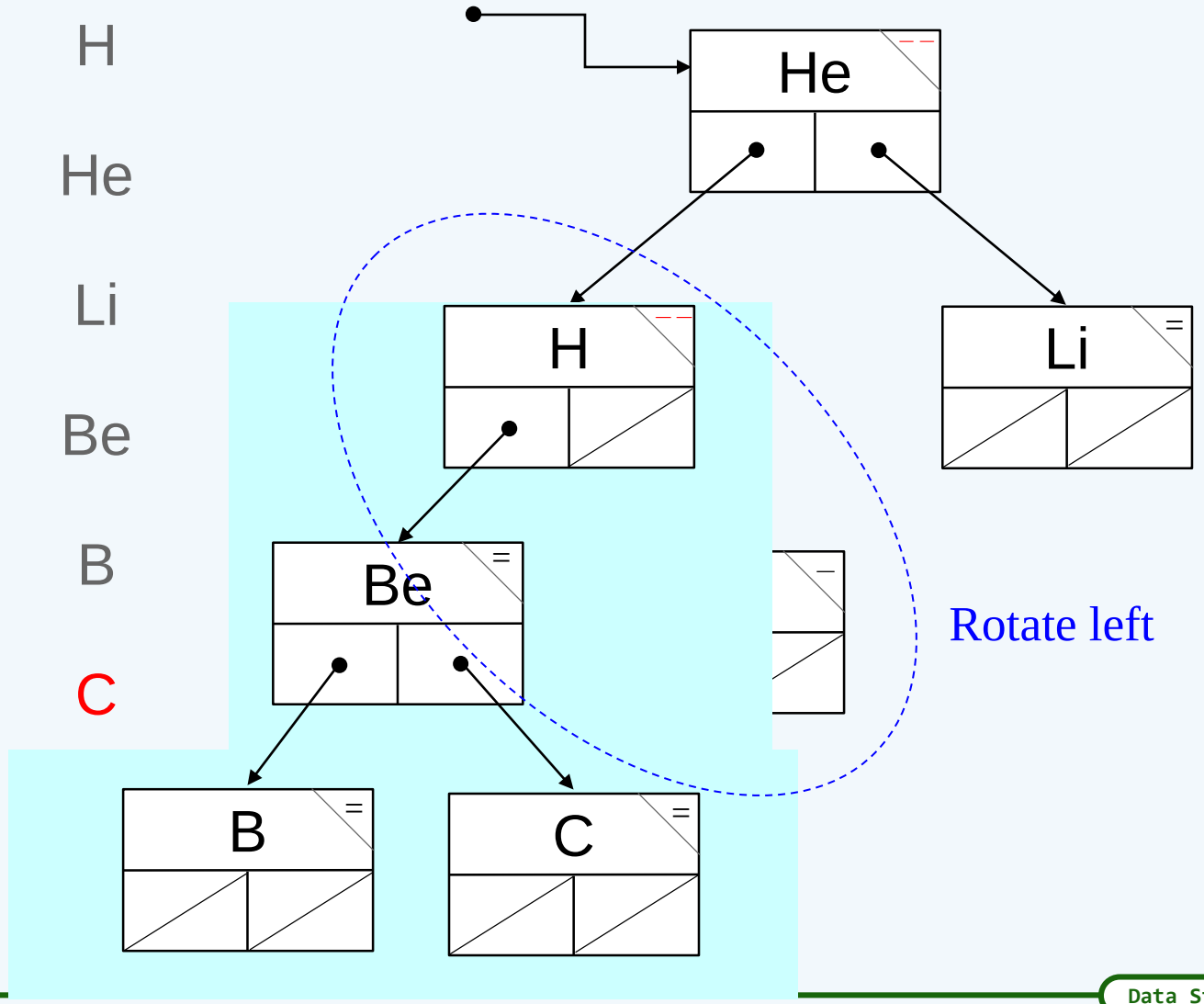
Be

B

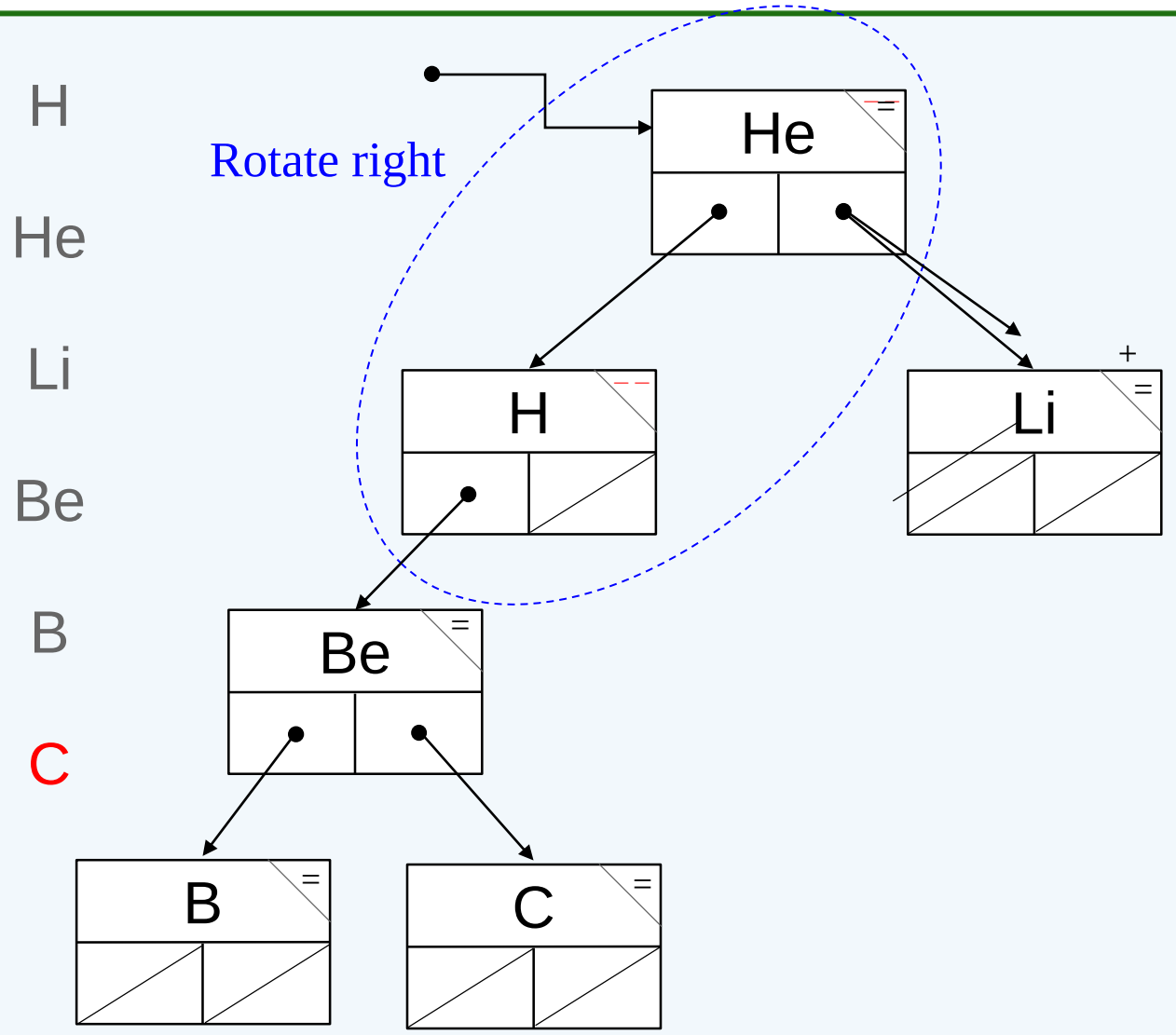
C



Illustrating the AVL Algorithm



Illustrating the AVL Algorithm



Tree-Balancing Algorithms

- The AVL algorithm was the first tree-balancing strategy and has been superseded by newer algorithms that are more effective in practice. These algorithms include:
  - Red-black trees
  - 2-3 trees
  - AA trees
  - Fibonacci trees
  - Splay trees

- In this course, the important thing to know is that it is *possible* to keep a binary tree balanced as you insert nodes, thereby ensuring that lookup operations run in $O(\log N)$ time. If you get really excited about this kind of algorithm, you'll have the opportunity to study them in more detail in the later course.

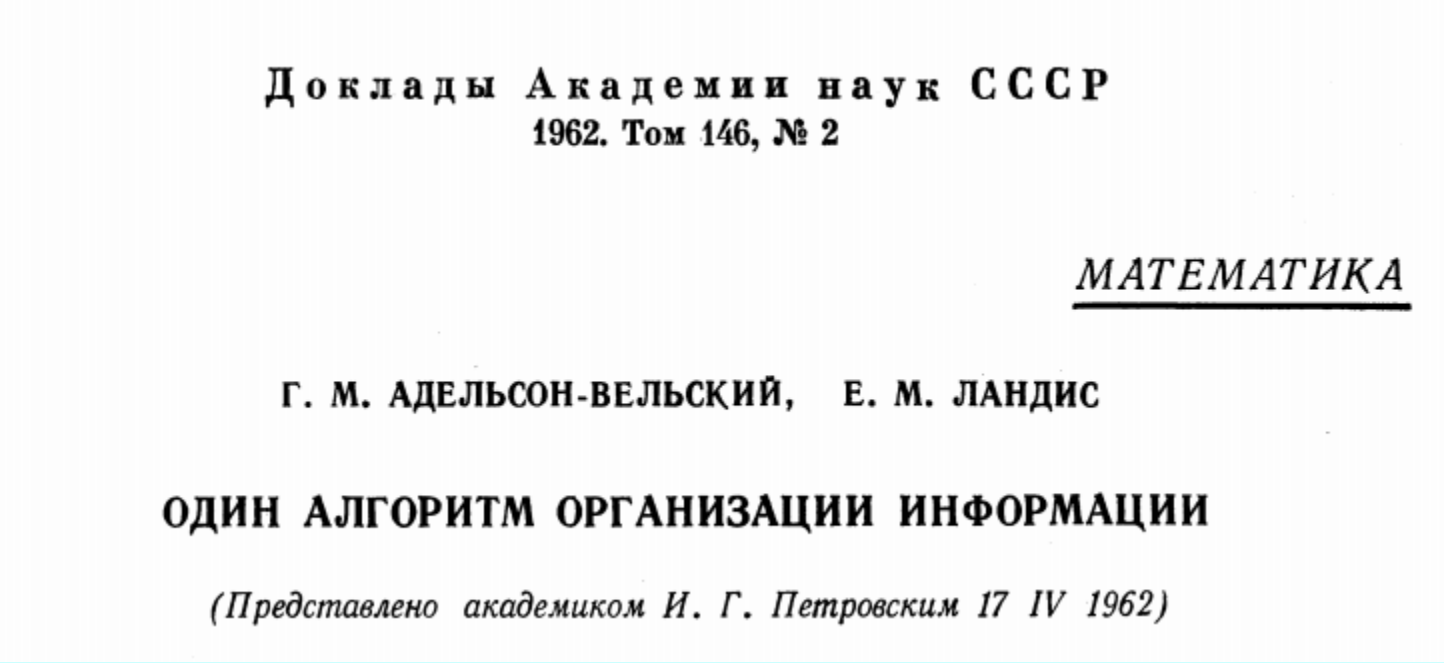
7.二叉搜索树

（c） AVL树

邓俊辉

deng@tsinghua.edu.cn

AVL Tree



Доклады Академии наук СССР
1962. Том 146, № 2

МАТЕМАТИКА

Г. М. АДЕЛЬСОН-ВЕЛЬСКИЙ, Е. М. ЛАНДИС

ОДИН АЛГОРИТМ ОРГАНИЗАЦИИ ИНФОРМАЦИИ

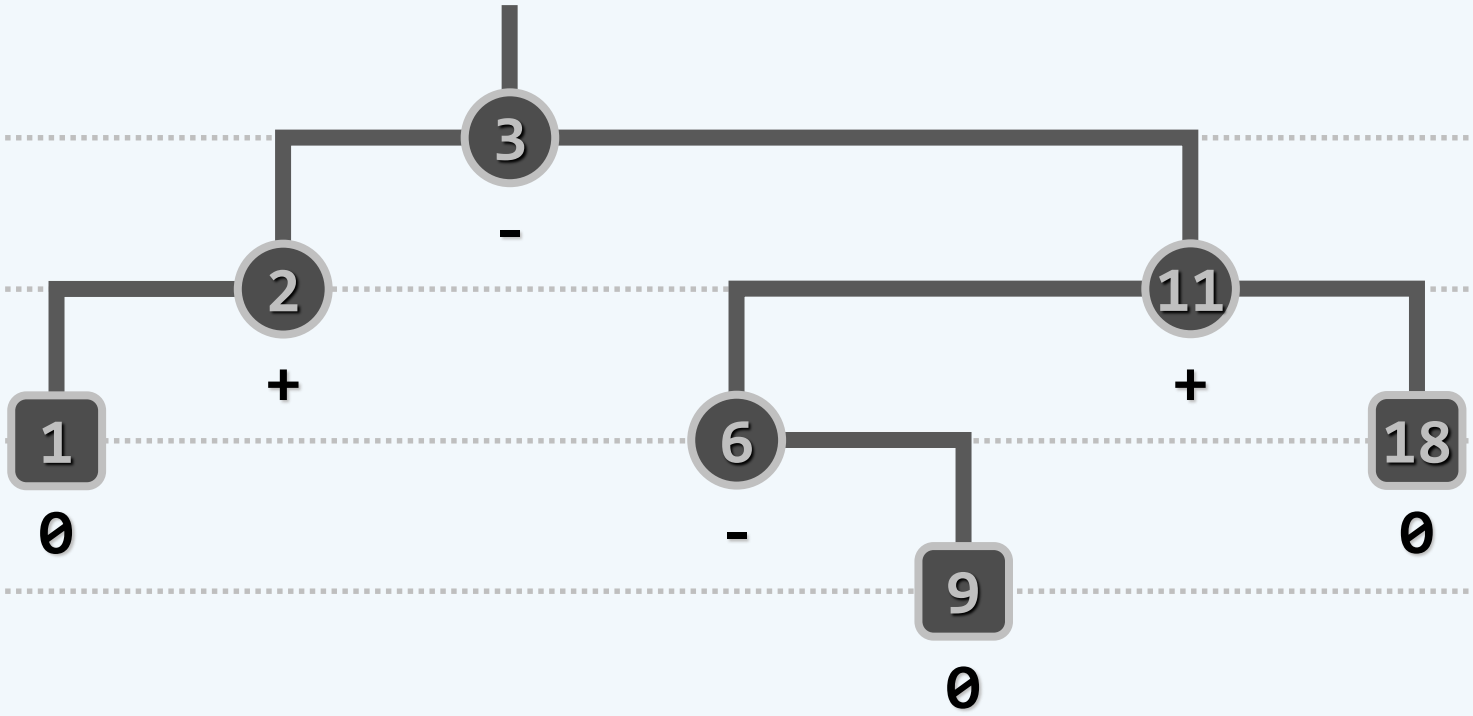
(Представлено академиком И. Г. Петровским 17 IV 1962)

G. Adelson-Velskii and E.M. Landis, "An algorithm for the organization of information." *Doklady Akademii Nauk SSSR*, 146:263–266, 1962 (Russian). English translation by Myron J. Ricci in *Soviet Math. Doklady*, 3:1259–1263, 1962.

平衡因子

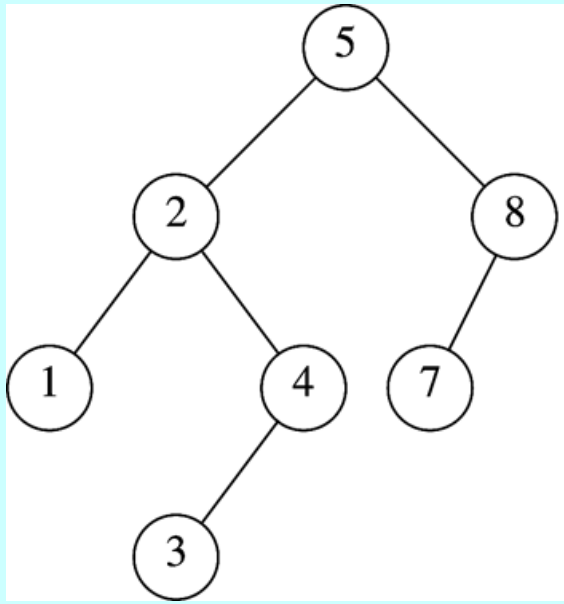
❖ balFac(v) = height(lc(v)) - height(rc(v))

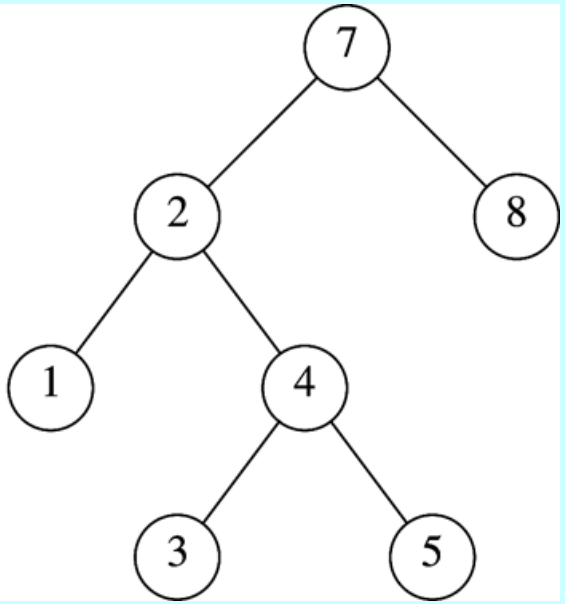
❖ G. Adelson-Velsky & E. Landis (1962): ∀ v, | balFac(v) | ≤ 1



❖ AVL树未必理想平衡，必然适度平衡...

Which is an AVL Tree?





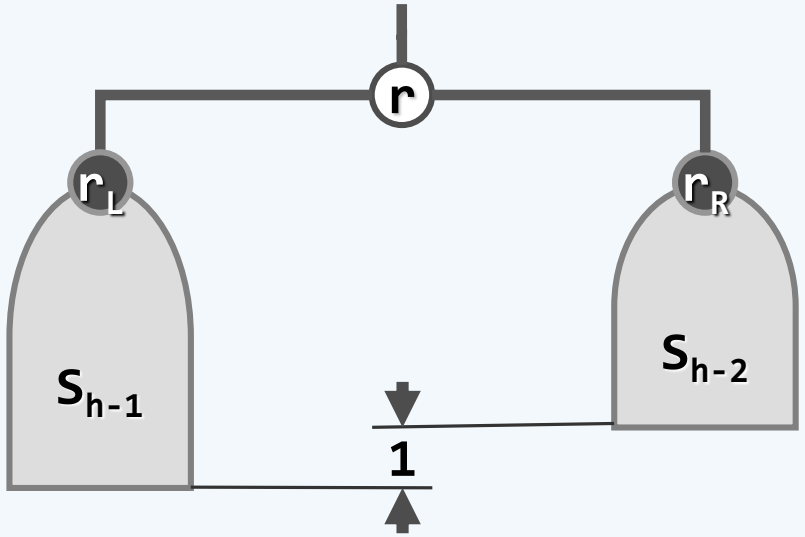
AVL = 适度平衡

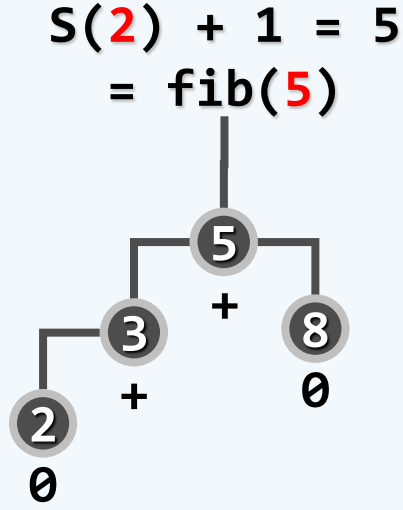
❖ 高度为h的AVL树，至少包含 S(h) = fib(h + 3) - 1 个节点

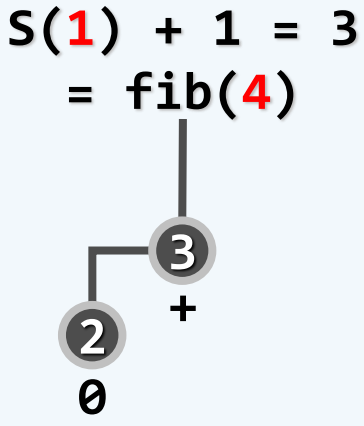
❖ S(h) = 1 + S(h - 1) + S(h - 2)

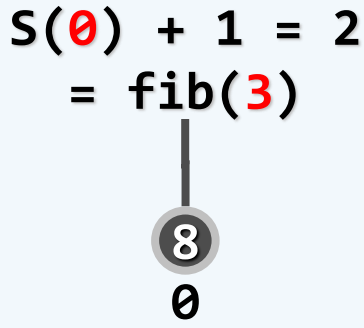
❖ S(h) + 1 = [S(h - 1) + 1] + [S(h - 2) + 1]

❖ fib(h + 3) = fib(h + 2) + fib(h + 1)









❖ 反过来，由n个节点构成的AVL树，高度至多为O(logn)

Height of an AVL Tree

Use Maple:

```
> rsolve( {F(0) = 1, F(1) = 2,
          F(h) = 1 + F(h - 1) + F(h - 2)}, F(h) );
```

$$\left(\frac{3\sqrt{5}}{10}+\frac{1}{2}\right)\left(\frac{1}{2}+\frac{\sqrt{5}}{2}\right)^{h}+\left(\frac{1}{2}-\frac{3\sqrt{5}}{10}\right)\left(\frac{1}{2}-\frac{\sqrt{5}}{2}\right)^{h}-\frac{2\sqrt{5}\left(-\frac{2}{1-\sqrt{5}}\right)^{h}}{5\left(1-\sqrt{5}\right)}+\frac{2\sqrt{5}\left(-\frac{2}{1+\sqrt{5}}\right)^{h}}{5\left(1+\sqrt{5}\right)}-1$$

```
> asympt( %, h );
```

$$\frac{\left(3\sqrt{5}+5\right)\left(1+\sqrt{5}\right)^{h}}{5\left(-1+\sqrt{5}\right)2^{h}}-1+\frac{1}{5}\frac{\mathbf{e}^{(h\,\pi\,I)}\left(-5+3\sqrt{5}\right)2^{h}}{\left(1+\sqrt{5}\right)\left(1+\sqrt{5}\right)^{h}}$$

$$c\left(\frac{1+\sqrt{5}}{2}\right)^{h}$$

Height of an AVL Tree

Алгоритм занесения таков, что в каждый момент справочный стол является допустимым деревом.

Лемма 1. *Пусть число ячеек допустимого дерева равно N. Тогда максимальная длина ветви не больше чем* $^3/_2 \log_2 (N+1)$.

Доказательство. Обозначим через N_n минимальное число ячеек в допустимом дереве при данной максимальной длине ветви n. Тогда легко доказывается (см. рис. 2), что $N_n = N_{n-1} + N_{n-2} + 1$.

Решая это уравнение в конечных разностях, получаем

$$N_n = \left(1 + \frac{2}{\sqrt{5}}\right)\left(\frac{1+\sqrt{5}}{2}\right)^n + \left(1 - \frac{2}{\sqrt{5}}\right)\left(\frac{1-\sqrt{5}}{2}\right)^n - 1.$$

Отсюда

$$n < \log_{\frac{1+\sqrt{5}}{2}} (N+1) < \frac{3}{2}\log_2 (N+1),$$

что и требовалось доказать.

Height of an AVL Tree

This is approximately

$$F(h) \approx 1.8944\ \phi^{h}$$

where $\phi \approx 1.6180$ is the golden ratio

– That is, $F(h) = \mathbf{O}(\phi^{h})$

Thus, we may find the maximum value of h for a given n:

$$\log_{\phi}(n / 1.8944) = \log_{\phi}(n) - 1.3277$$

Height of an AVL Tree

If $n = 10^6$, the bounds on h are:

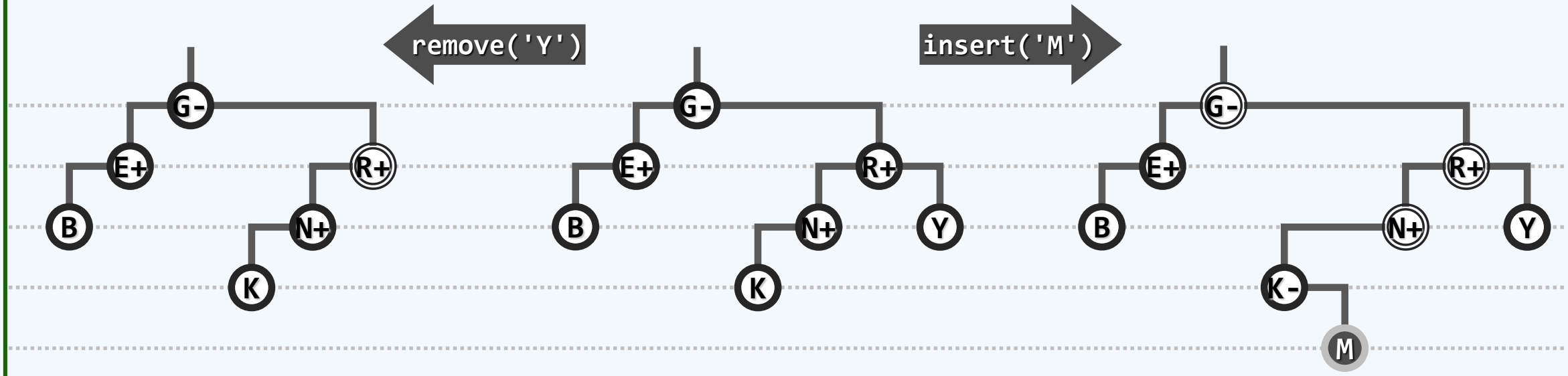
- The minimum height: $\log_2(10^6) - 1 \approx 19$
- the maximum height : $\log_\phi(10^6 / 1.8944) < 28$

AVL：接口

```
❖ #define Balanced(x) \ //理想平衡
     ( stature( (x).lc ) == stature( (x).rc ) )
  #define BalFac(x) \ //平衡因子
     ( stature( (x).lc ) - stature( (x).rc ) )
  #define AvlBalanced(x) \ //AVL平衡条件
     ( ( -2 < BalFac(x) ) && ( BalFac(x) < 2 ) )
❖ template <typename T> class AVL : public BST<T> { //由BST派生
  public: // BST::search()等接口，可直接沿用
     BinNodePosi(T) insert( const T & ); //插入重写
     bool remove( const T & ); //删除重写
  };
```

失衡与重平衡

❖ 按BST规则插入或删除节点之后，AVL平衡性可能破坏——如何恢复?



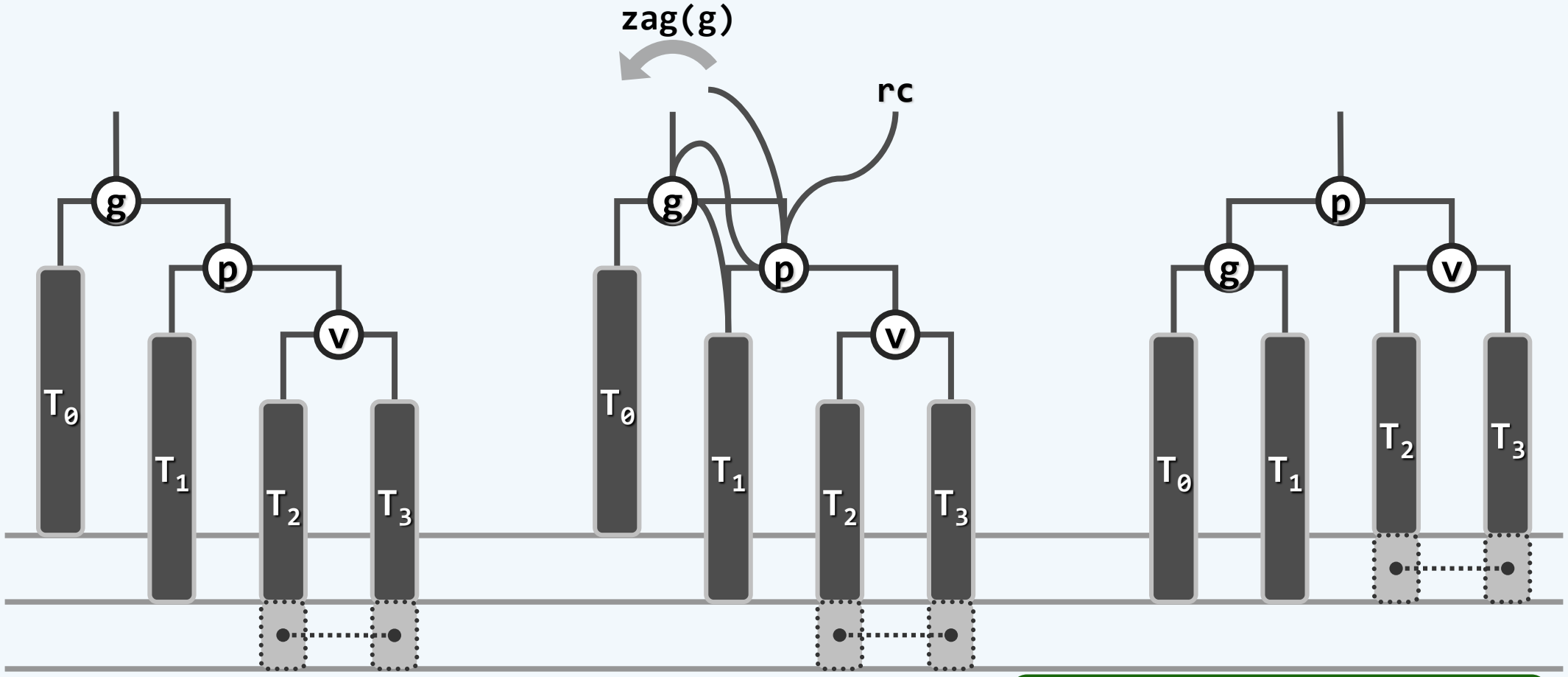
❖ 蛮力不足取，须借助等价变换

局部性：所有的旋转都在局部进行 //每次只需O(1)时间

快速性：在每一深度只需检查并旋转至多一次 //共O(logn)次

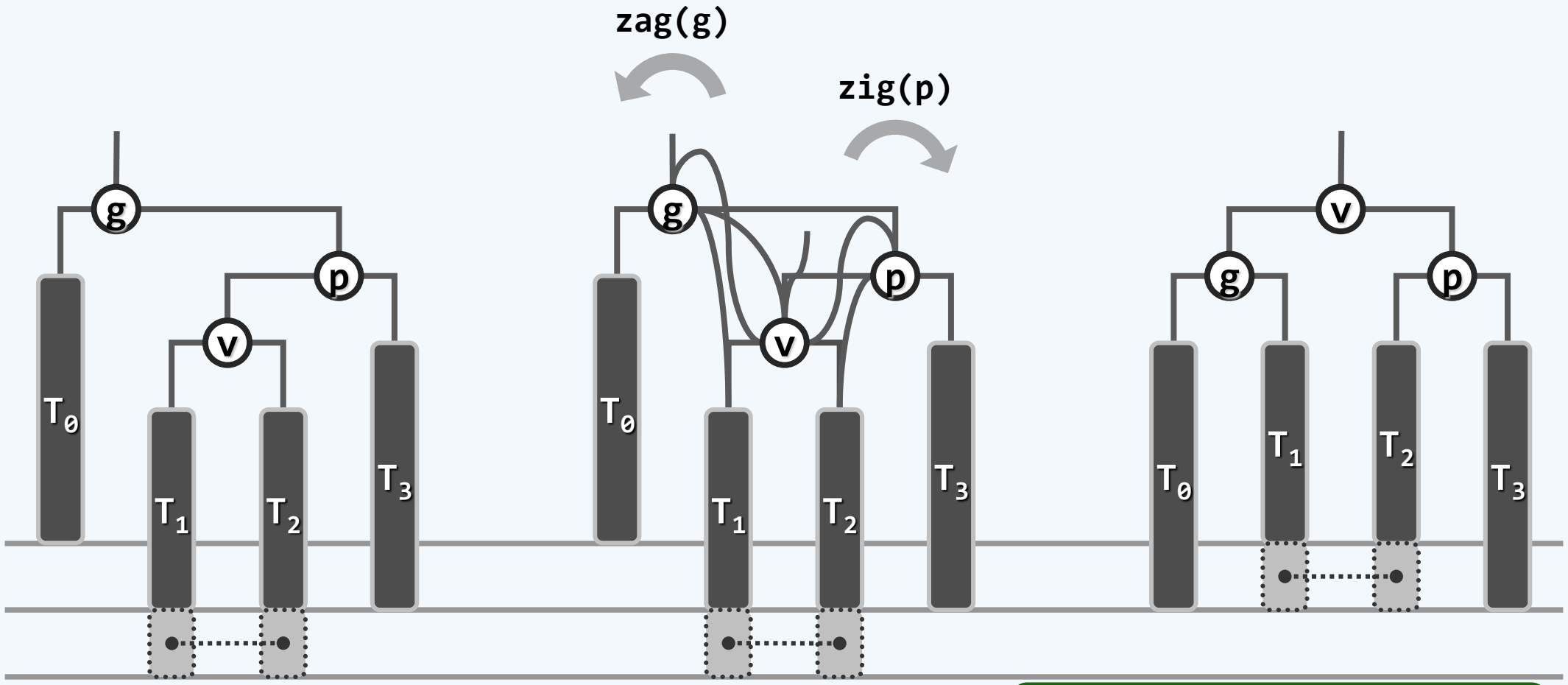
插入：单旋

- 同时可有多个失衡节点，最低者g不低于x祖父
- g经单旋调整后复衡，子树高度复原；更高祖先也必平衡，全树复衡



插入：双旋

- 同时可有多个失衡节点，最低者g不低于x祖父
- g经双旋调整后复衡，子树高度复原；更高祖先也必平衡，全树复衡

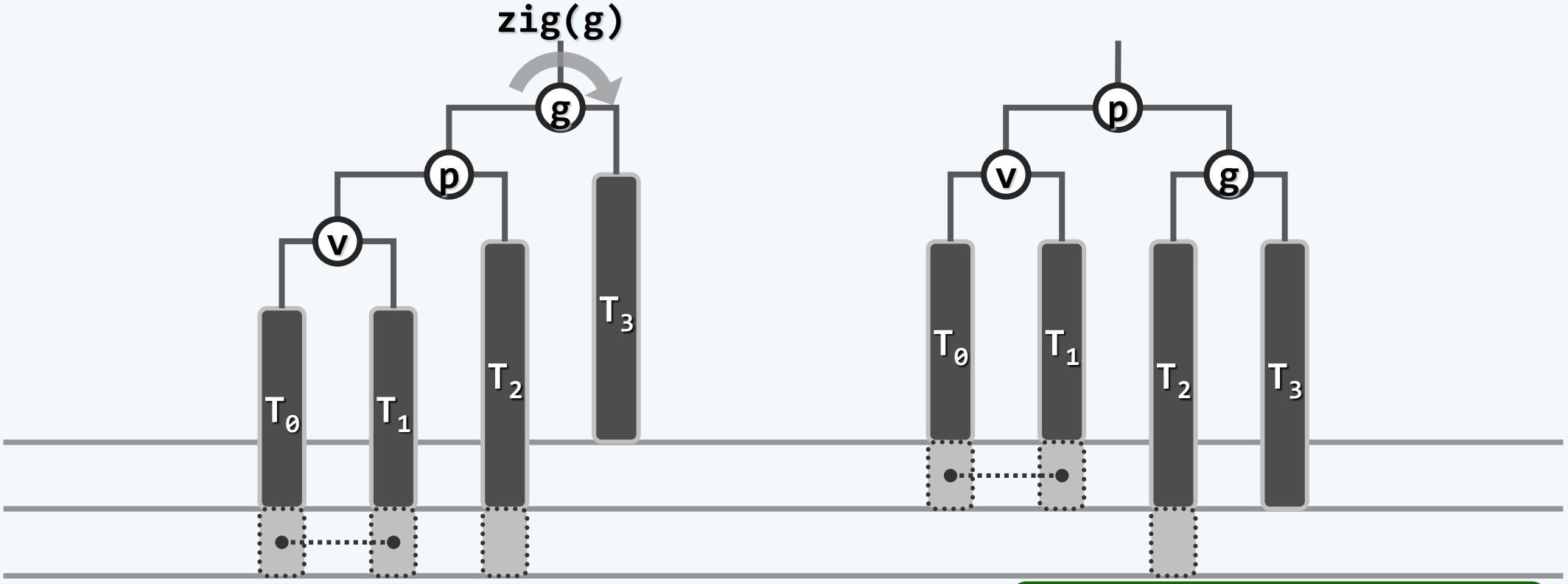


插入：实现

```
❖template <typename T> BinNodePosi(T) AVL<T>::insert( const T & e ) {
    BinNodePosi(T) & x = search( e ); if ( x ) return x; //若目标尚不存在
    BinNodePosi(T) xx = x = new BinNode<T>( e, _hot ); _size++; //则创建新节点
  // 此时，若x的父亲_hot增高，则祖父有可能失衡。故以下从_hot起，向上逐层检查各代祖先
    for ( BinNodePosi(T) g = _hot; g; g = g->parent )
       if ( ! AvlBalanced( *g ) ) { //一旦发现g失衡，则通过调整恢复平衡
          FromParentTo( *g ) = rotateAt( tallerChild( tallerChild( g ) ) );
          break; //g复衡后，局部子树高度必然复原；其祖先亦必如此，故调整结束
       } else //否则（在依然平衡的祖先处），只需简单地
          updateHeight( g ); //更新其高度（平衡性虽不变，高度却可能改变）
    return xx; //返回新节点：至多只需一次调整
  }
```

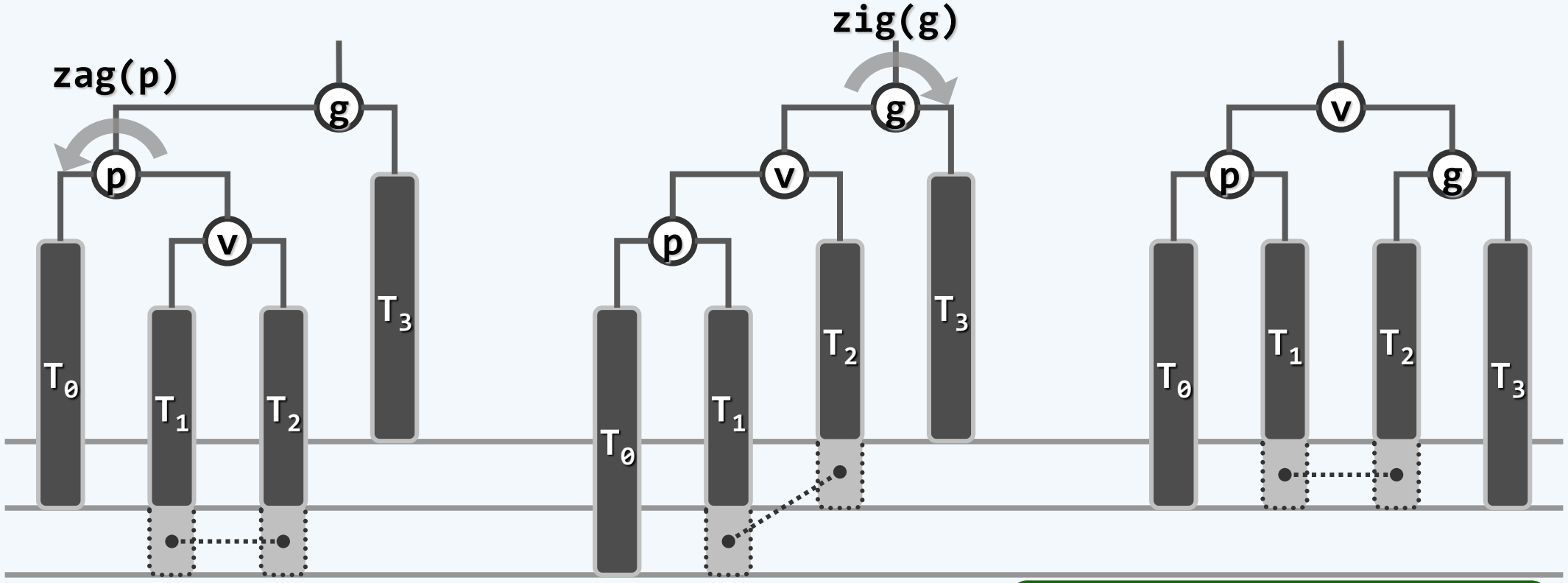
删除：单旋

- 同时至多一个失衡节点g，首个可能就是x的父亲_hot
- g经单旋调整后复衡，子树高度未必复原；更高祖先仍可能失衡
- 因有失衡传播现象，可能需做O(logn)次调整



删除：双旋

- 同时至多一个失衡节点g，首个可能就是x的父亲_hot
- g经单旋调整后复衡，子树高度未必复原；更高祖先仍可能失衡
- 因有失衡传播现象，可能需做O(logn)次调整



删除：实现

```
❖template <typename T> bool AVL<T>::remove( const T & e ) {
    BinNodePosi(T) & x = search( e ); if ( !x ) return false; //若目标的确存在
    removeAt( x, _hot ); _size--; //则在按BST规则删除之后，_hot及祖先均有可能失衡
  // 以下，从_hot出发逐层向上，依次检查各代祖先g
    for ( BinNodePosi(T) g = _hot; g; g = g->parent ) {
       if ( ! AvlBalanced( *g ) ) //一旦发现g失衡，则通过调整恢复平衡
          g = FromParentTo( *g ) = rotateAt( tallerChild( tallerChild( g ) ) );
       updateHeight( g ); //并更新其高度
    } //可能需做过Ω(logn)次调整；无论是否做过调整，全树高度均可能下降
    return true; //删除成功
  }
```

3+4重构：算法

❖ 设g(x)为最低的失衡节点，考察祖孙三代：`g ~ p ~ v`

按**中序**遍历次序，将其**重命名**为：`a < b < c`

❖ 它们总共拥有互不相交的四棵（可能为空的）子树

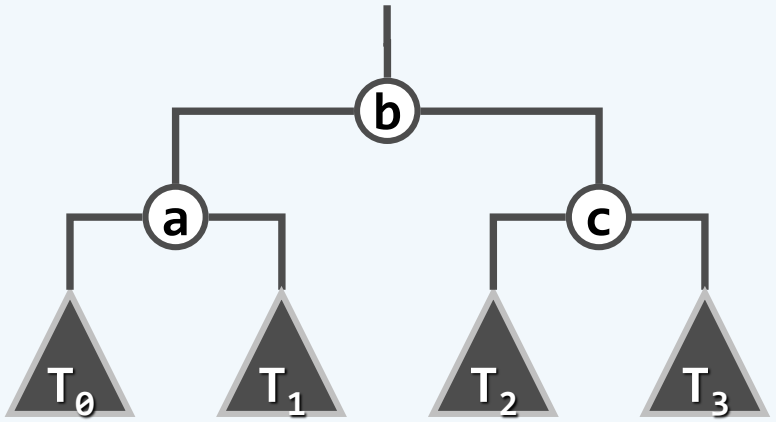
按**中序**遍历次序，将其重命名为：$T_0 < T_1 < T_2 < T_3$

❖ 将原先以g为根的子树S，替换为一棵新子树S'

`root(S') = b`

`lc(b) = a`　　`rc(b) = c`

`lT(a) = T0`　`rT(a) = T1`　`lT(c) = T2`　`rT(c) = T3`



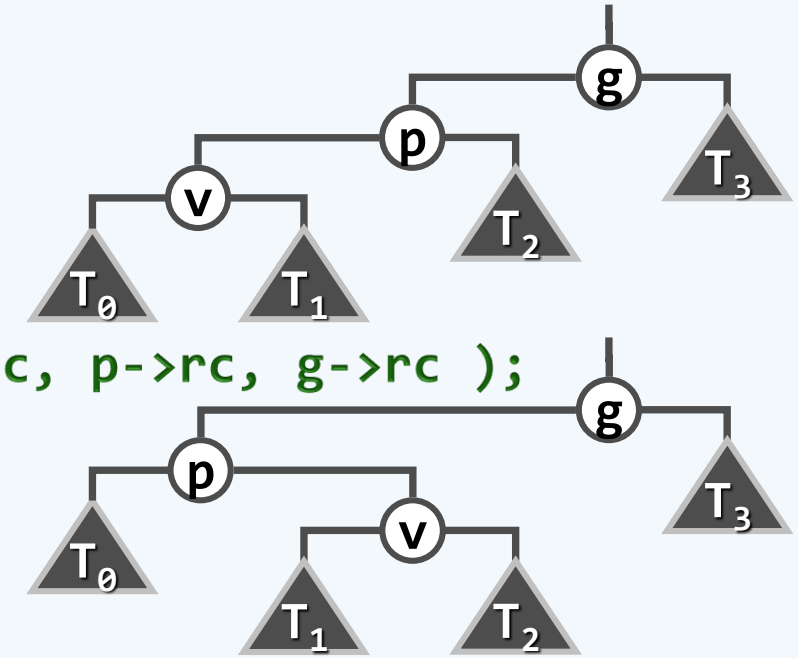
❖ 等价变换，保持中序遍历次序：$T_0 < a < T_1 < b < T_2 < c < T_3$

3+4重构：实现

```
❖template <typename T> BinNodePosi(T) BST<T>::connect34(
    BinNodePosi(T) a, BinNodePosi(T) b, BinNodePosi(T) c,
    BinNodePosi(T) T0, BinNodePosi(T) T1, BinNodePosi(T) T2, BinNodePosi(T) T3)
  {
    a->lc = T0; if (T0) T0->parent = a;
    a->rc = T1; if (T1) T1->parent = a; updateHeight(a);
    c->lc = T2; if (T2) T2->parent = c;
    c->rc = T3; if (T3) T3->parent = c; updateHeight(c);
    b->lc = a; a->parent = b;
    b->rc = c; c->parent = b; updateHeight(b);
    return b; //该子树新的根节点
  }
```

统一调整：实现

```
❖ template<typename T> BinNodePosi(T) BST<T>::rotateAt( BinNodePosi(T) v ) {
    BinNodePosi(T) p = v->parent, g = p->parent; //父亲、祖父
    if ( IsLChild( * p ) ) //zig
       if ( IsLChild( * v ) ) { //zig-zig
          p->parent = g->parent; //向上联接
          return connect34( v, p, g, v->lc, v->rc, p->rc, g->rc );
       } else { //zig-zag
          v->parent = g->parent; //向上联接
          return connect34( p, v, g, p->lc, v->lc, v->rc, g->rc );
       }
    else { /*.. zag-zig & zag-zag ..*/ }
 }
```



综合评价

❖ 优点　无论查找、插入或删除，最坏情况下的复杂度均为O(logn)

O(n)的存储空间

❖ 缺点　借助高度或平衡因子，为此需改造元素结构，或额外封装

实测复杂度与理论值尚有差距

插入/删除后的旋转，成本不菲

删除操作后，最多需旋转Ω(logn)次（Knuth：平均仅0.21次）

若需频繁进行插入/删除操作，未免得不偿失

单次动态调整后，全树拓扑结构的变化量可能高达Ω(logn)

❖ 有没有更好的结构呢？　//保持兴趣

Reference

- *http://www.cs.usfca.edu/~galles/visualization/Algorithms.html*
- 数据结构(C++语言版)第三版 Chapter 7
- *Data Structures and Algorithm Analysis in C++* Section 4.4
- http://users.cis.fiu.edu/~weiss/dsaa_c++3/code/AvlTree.h

Next

- Splay Tree
- 数据结构（C++语言版）第三版 Chapter 8.1