高级搜索树
伸展树：逐层伸展

局部性

❖ Locality：刚被访问过的数据，极有可能很快地再次被访问

这一现象在信息处理过程中屡见不鲜...

[Pfaff-2004] Performance Analysis Of BSTs In System Software

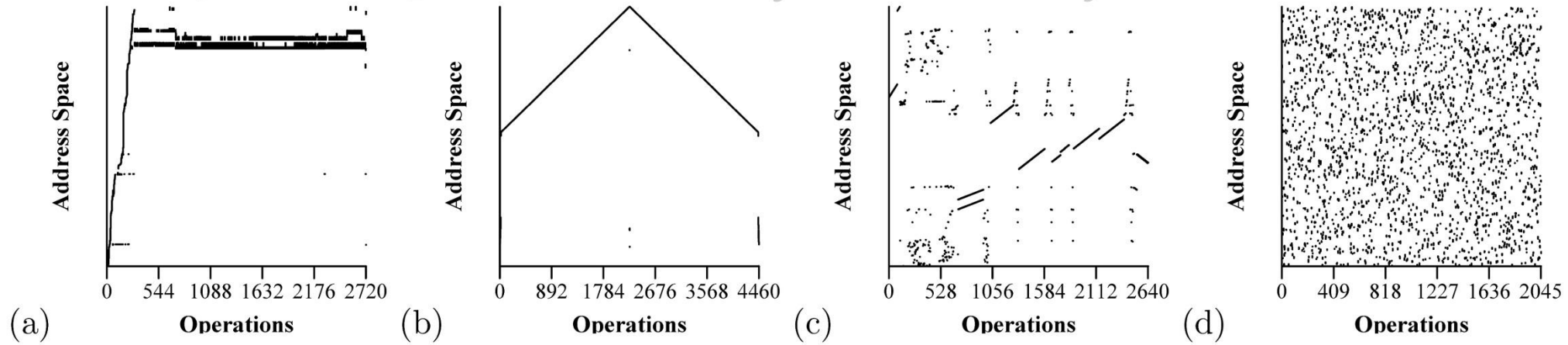


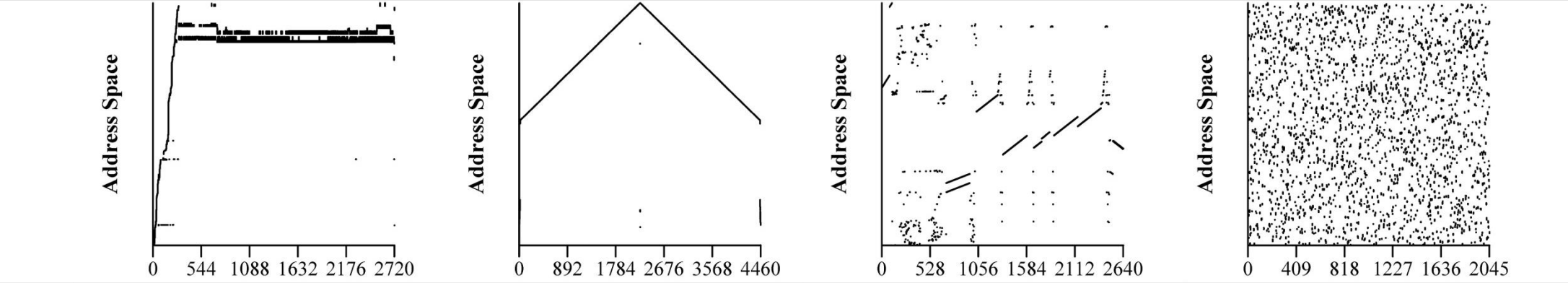
Figure 2: Call sequences in (a) Mozilla 1.0, (b) VMware GSX Server 2.0.1, (c) squid running under User-Mode Linux 2.4.18.48, and (d) random test sets. Part (b) omits one `mmap-munmap` pair for memory region 0x20000000 to 0x30000000 and (c) omits address space gaps; the others are complete.

局部性

- BST：　刚被访问过的节点，极有可能很快地再次被访问

  下一将要访问的节点，极有可能就在刚被访问过节点的附近

- 对AVL连续的m次查找（m >> n），共需 $\mathcal{O}(m \cdot \log n)$ 时间——能否利用局部性加速?



- 自适应链表：节点一旦被访问，随即移动到最前端

  模仿：BST的节点一旦被访问，随即调整到树根

- 难点：如何实现这种调整？调整过程自身的复杂度如何控制?

逐层伸展

- 节点v一旦被访问

  随即被推送至根

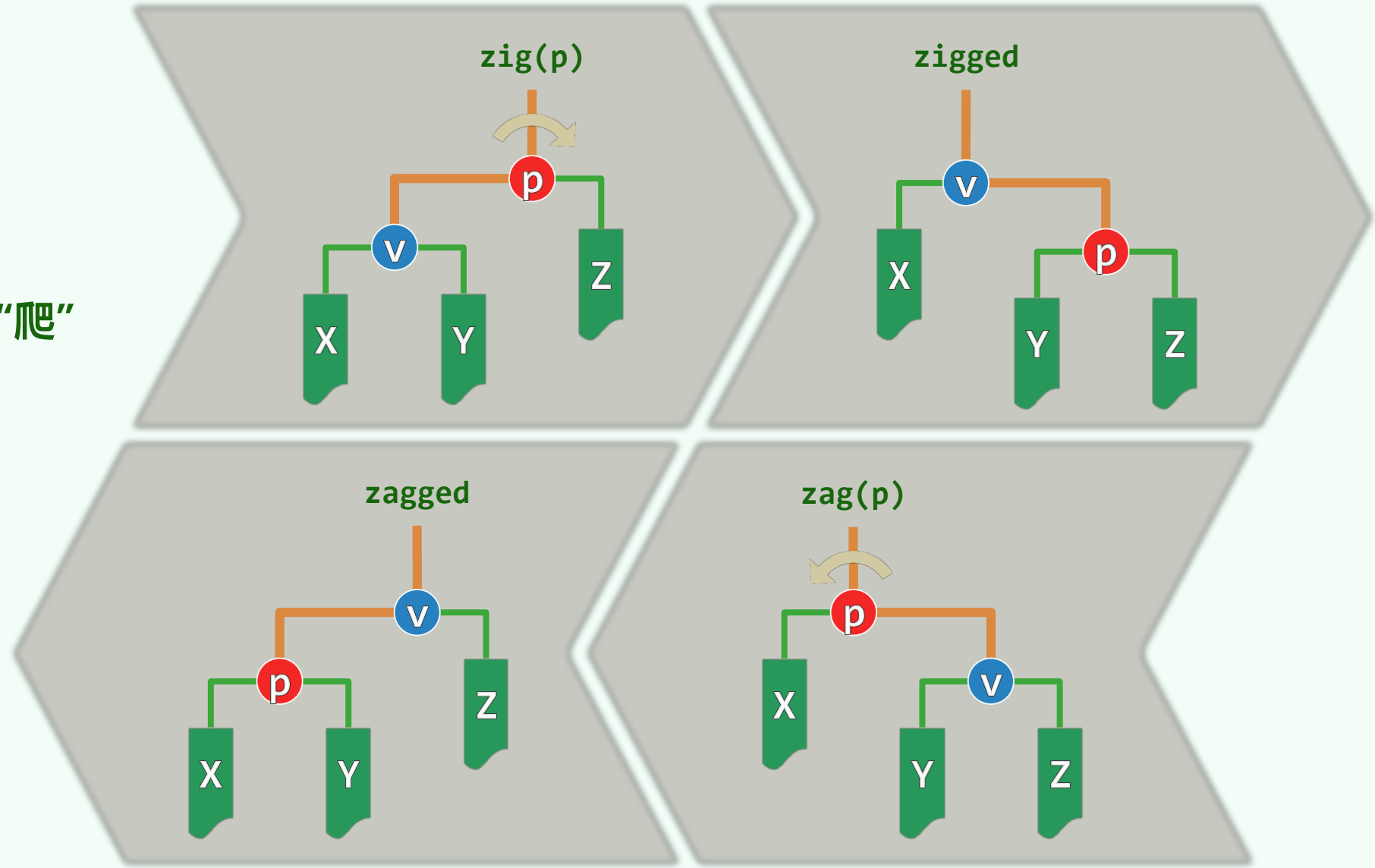
- 与其说“推”，不如说“爬”

  一步一步地往上爬

- 自下而上，逐层旋转

  - zig(v->parent)

  - zag(v->parent)



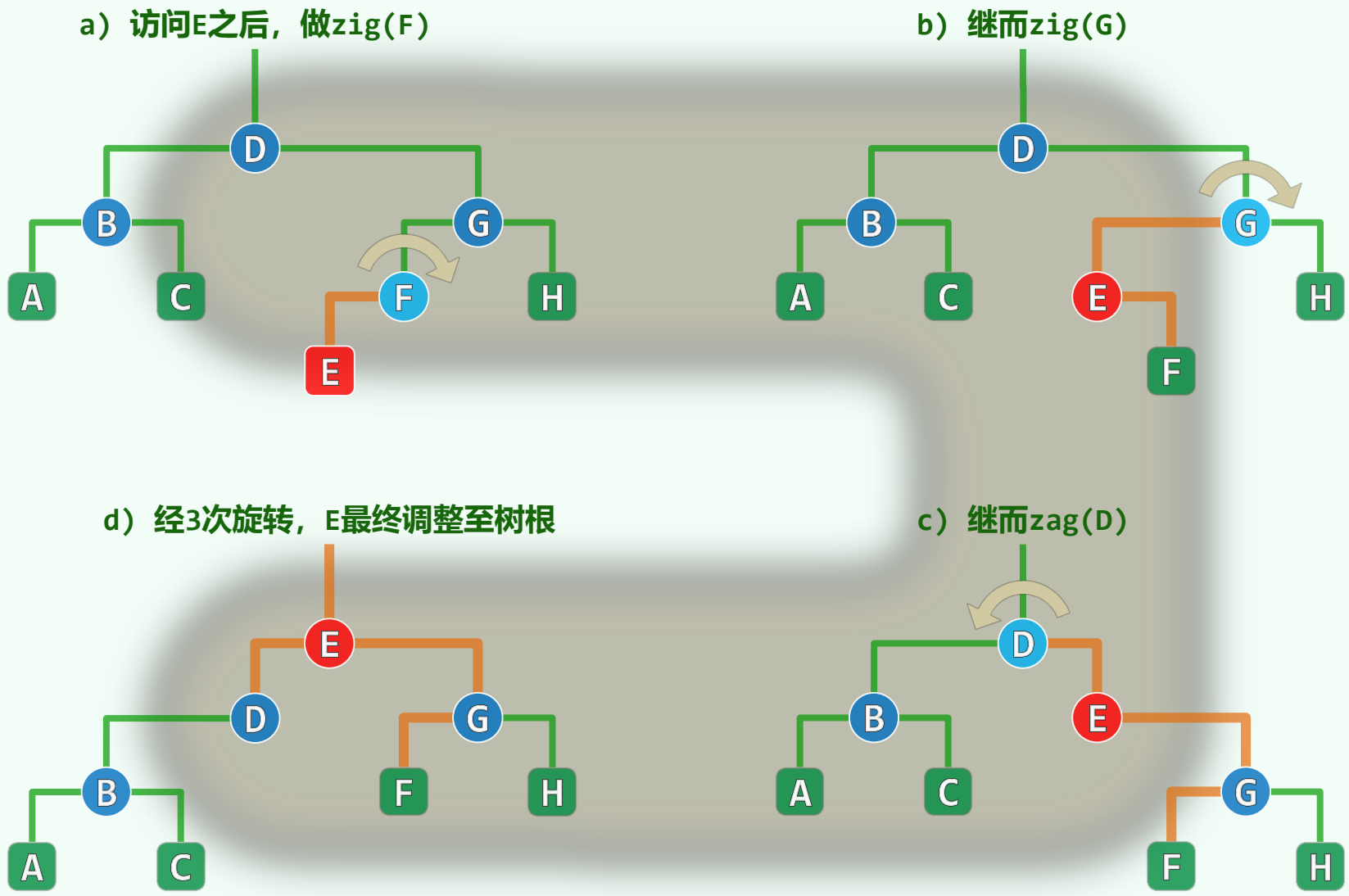
实例

- 伸展过程的效率

  是否足够地高?

- 这取决于

  - 树的初始形态和
  - 节点的访问次序



最坏情况

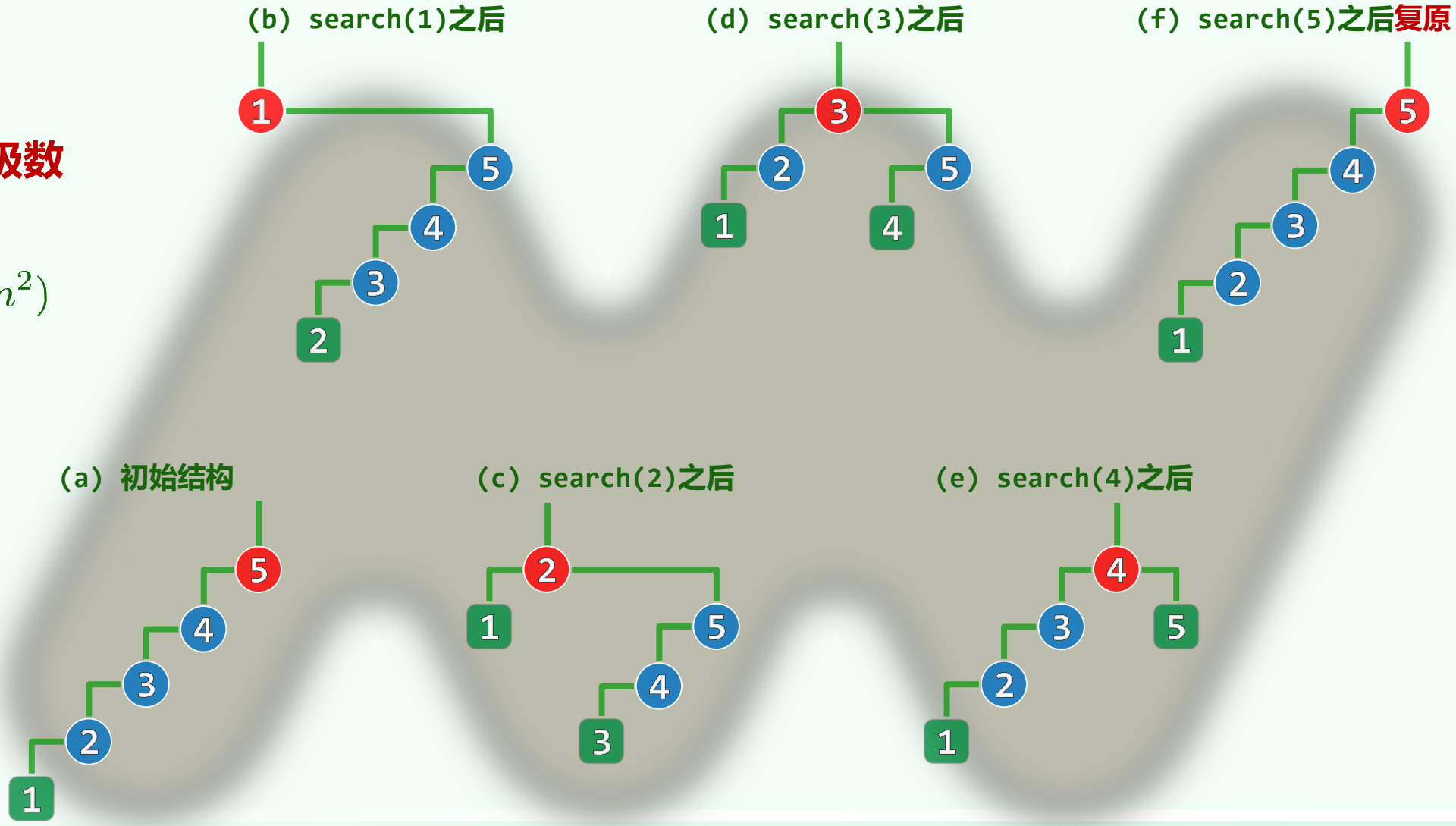
- 旋转次数

  呈周期性的算术级数

- 每一周期累计$\Omega(n^2)$

  分摊 $\Omega(n)$

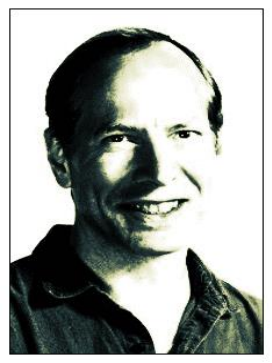
- 怎么破?



高级搜索树
伸展树：双层伸展

双层伸展

❖ Self-Adjusting Binary Trees

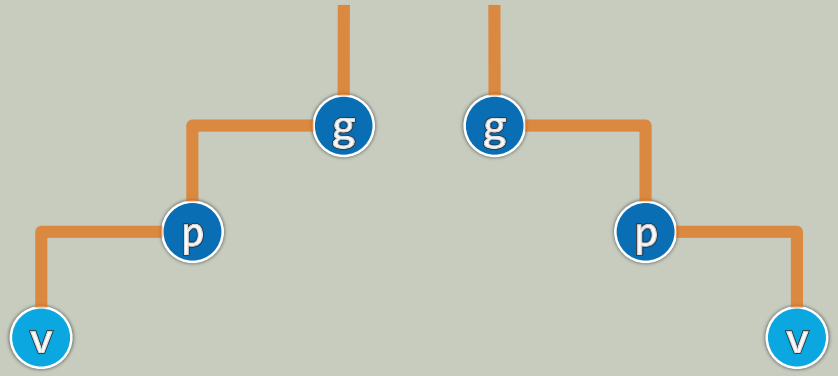


D. D. Sleator

R. E. Tarjan

J. ACM, 32:652-686, 1985

❖ 构思的精髓：向上追溯两层，而非一层



❖ 反复考察祖孙三代：

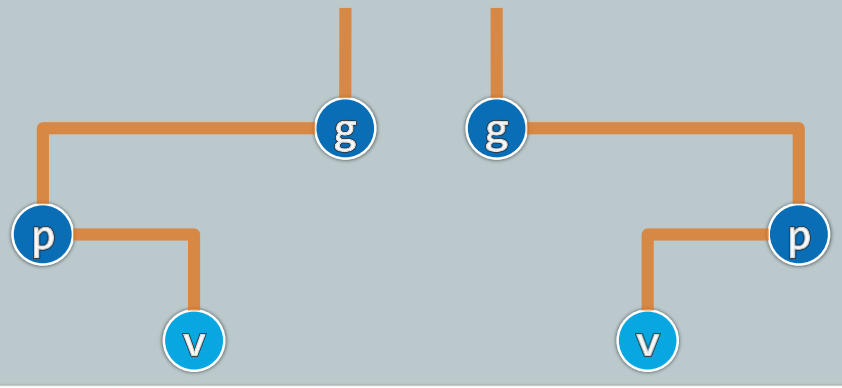
g = parent(p), p = parent(v), v

❖ 根据它们的相对位置，经两次旋转

使v上升两层，成为（子）树根

❖ 如此，性能的确会有改善?

❖ 具体地，应该如何旋转?



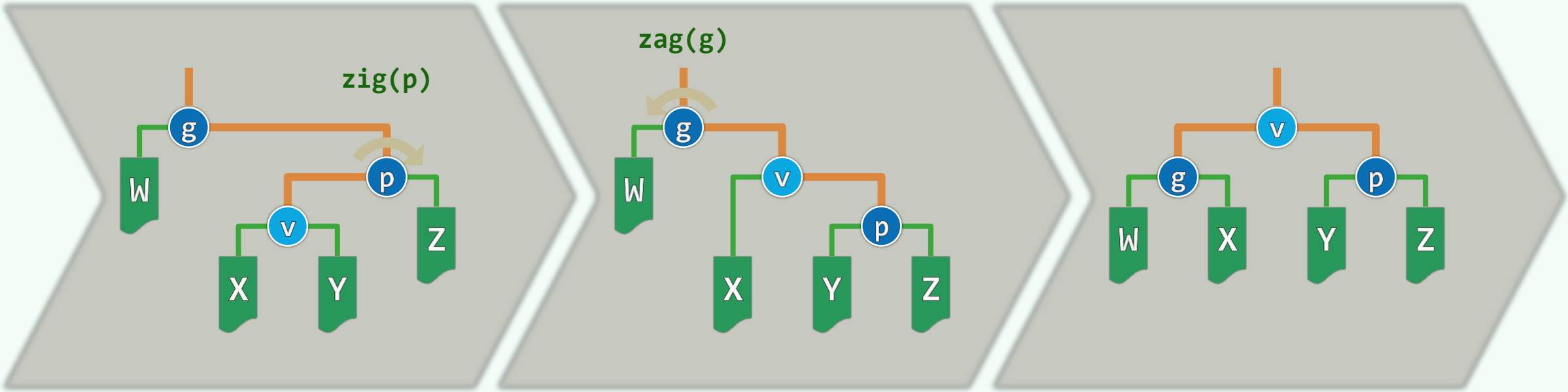
zig-zag / zag-zig

❖ 此时的v按中序遍历次序居中

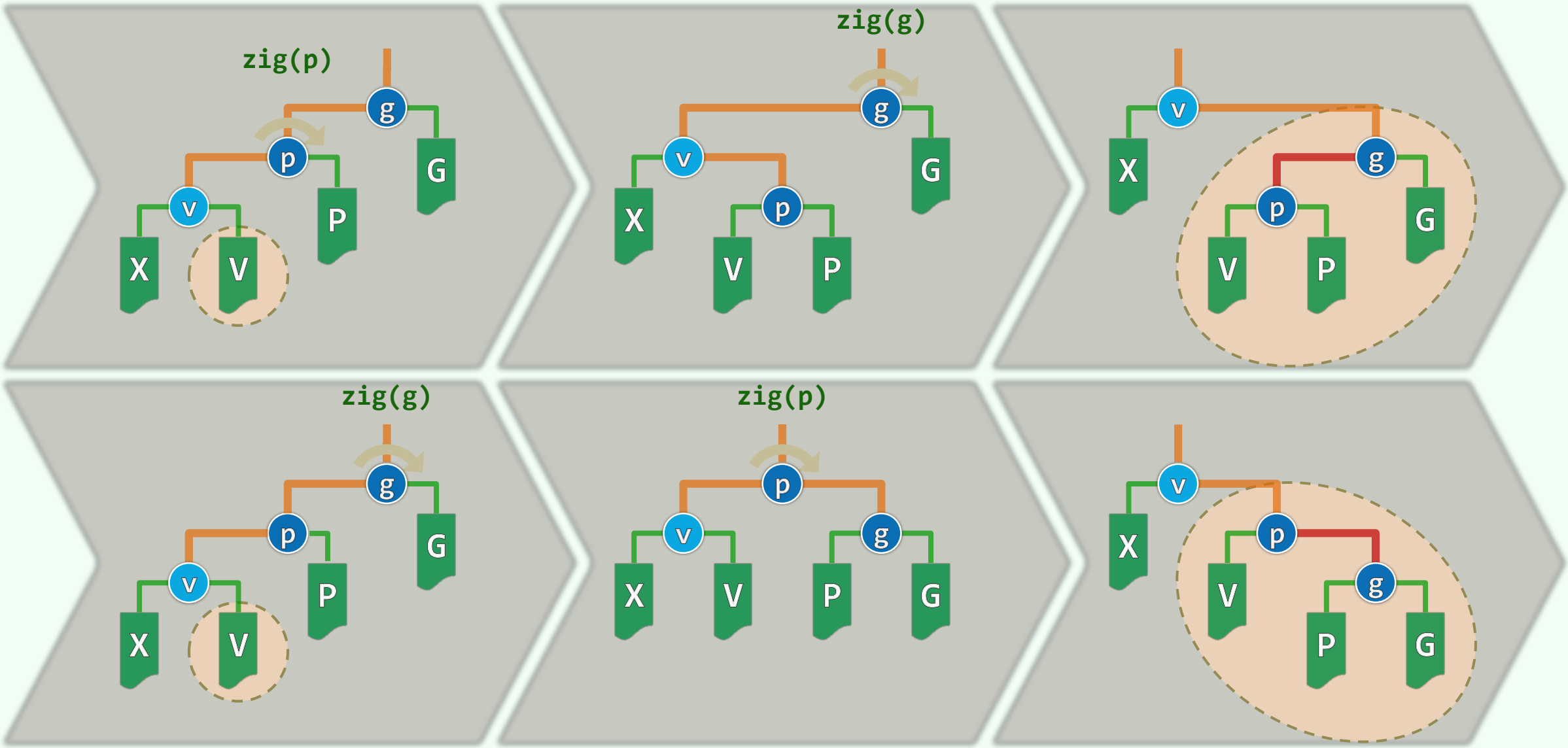
❖ 如此调整的效果，与逐层调整别无二致！

❖ 故若欲使之成为根，最终无非一种姿势

❖ 难道，就这样平淡无奇？



zig-zig / zag-zag



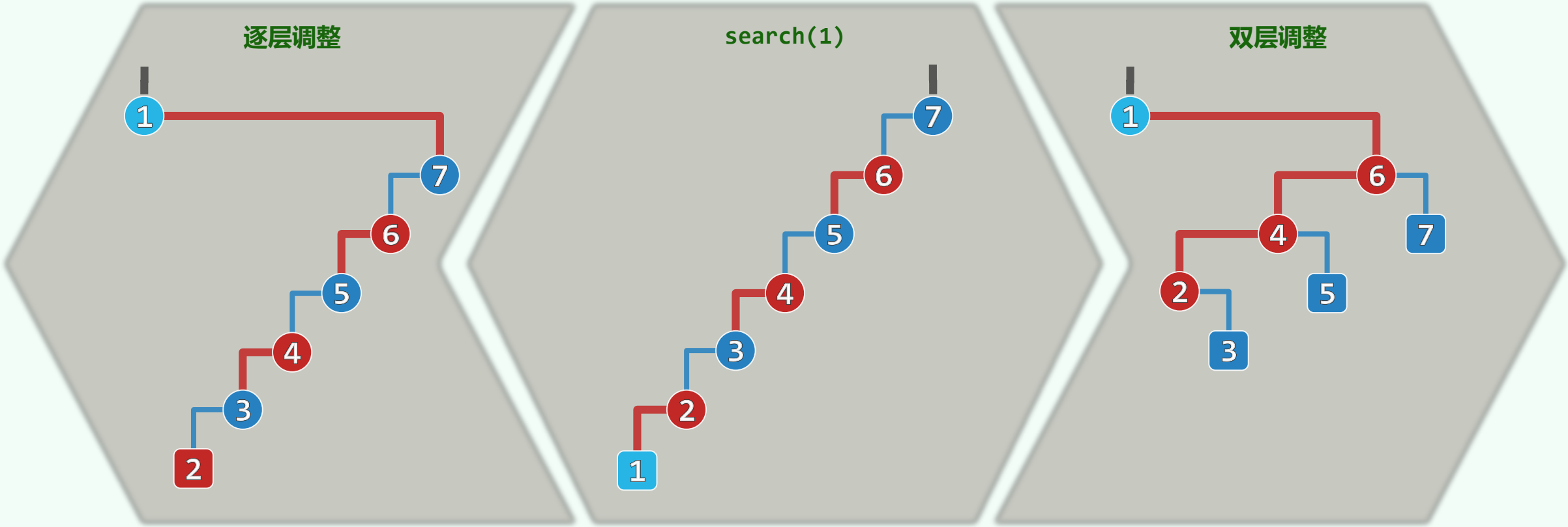
zig-zig / zag-zag

❖ 节点访问之后，对应路径的长度随即折半

//含羞草般的折叠效果

❖ 最坏情况不致持续发生！

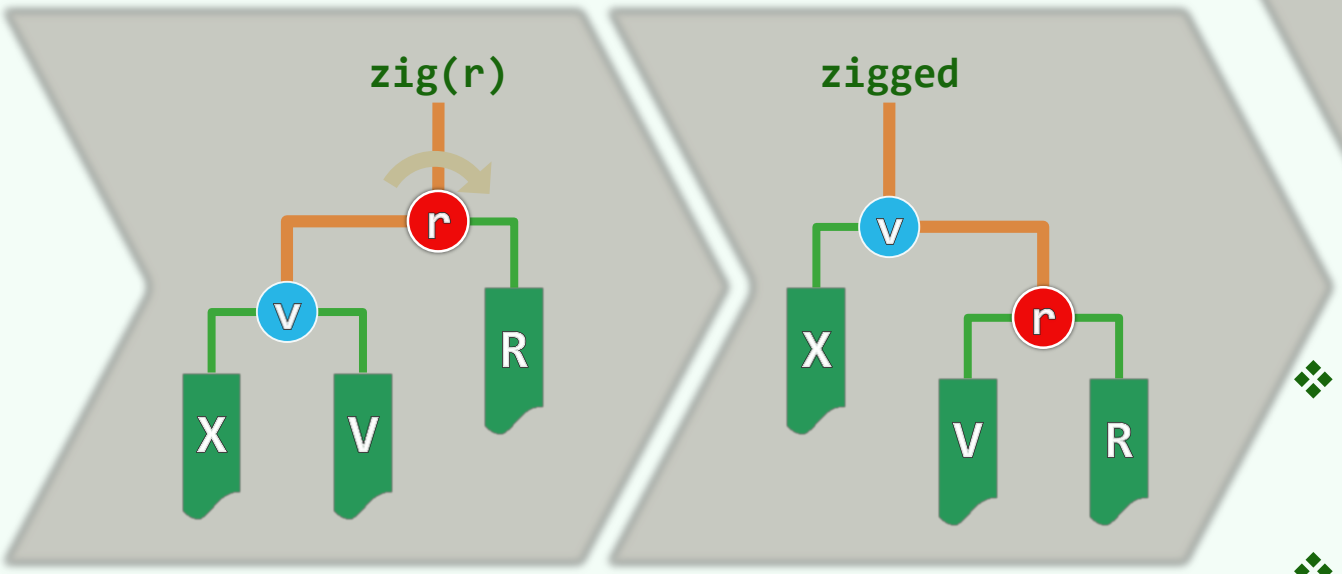
习题[8-2]：伸展操作分摊仅需 $\mathcal{O}(\log n)$ 时间

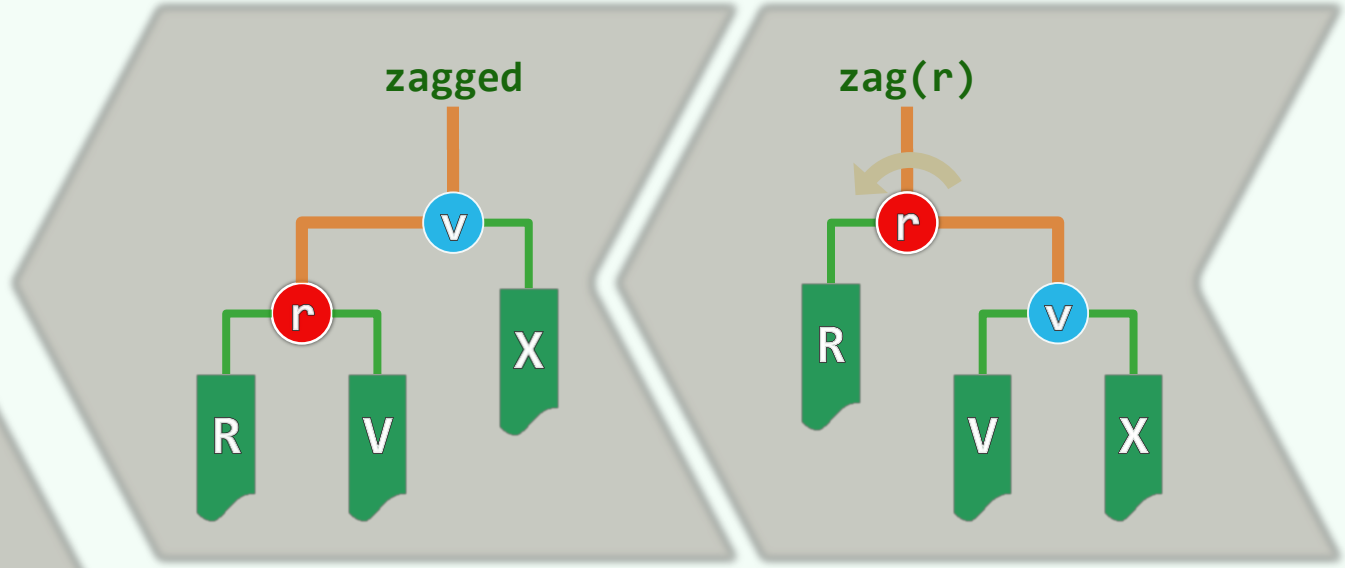


zig / zag

❖ 要是v只有父亲，没有祖父呢?

❖ 此时必有v.parent() == T.root()





❖ 只需做单次旋转：zig(r)或zag(r)

❖ 好在，这种情况至多（在最后）出现一次

高级搜索树
伸展树：算法实现

接口

```
template <typename T> class Splay : public BST<T> { //由BST派生
   protected:
      BinNodePosi<T> splay( BinNodePosi<T> v ); //将v伸展至根
   public: //伸展树的查找也会引起整树的结构调整，故search()也需重写
      BinNodePosi<T> & search( const T & e ); //查找（重写）
      BinNodePosi<T> insert( const T & e ); //插入（重写）
      bool remove( const T & e ); //删除（重写）
};
```

伸展算法：总体

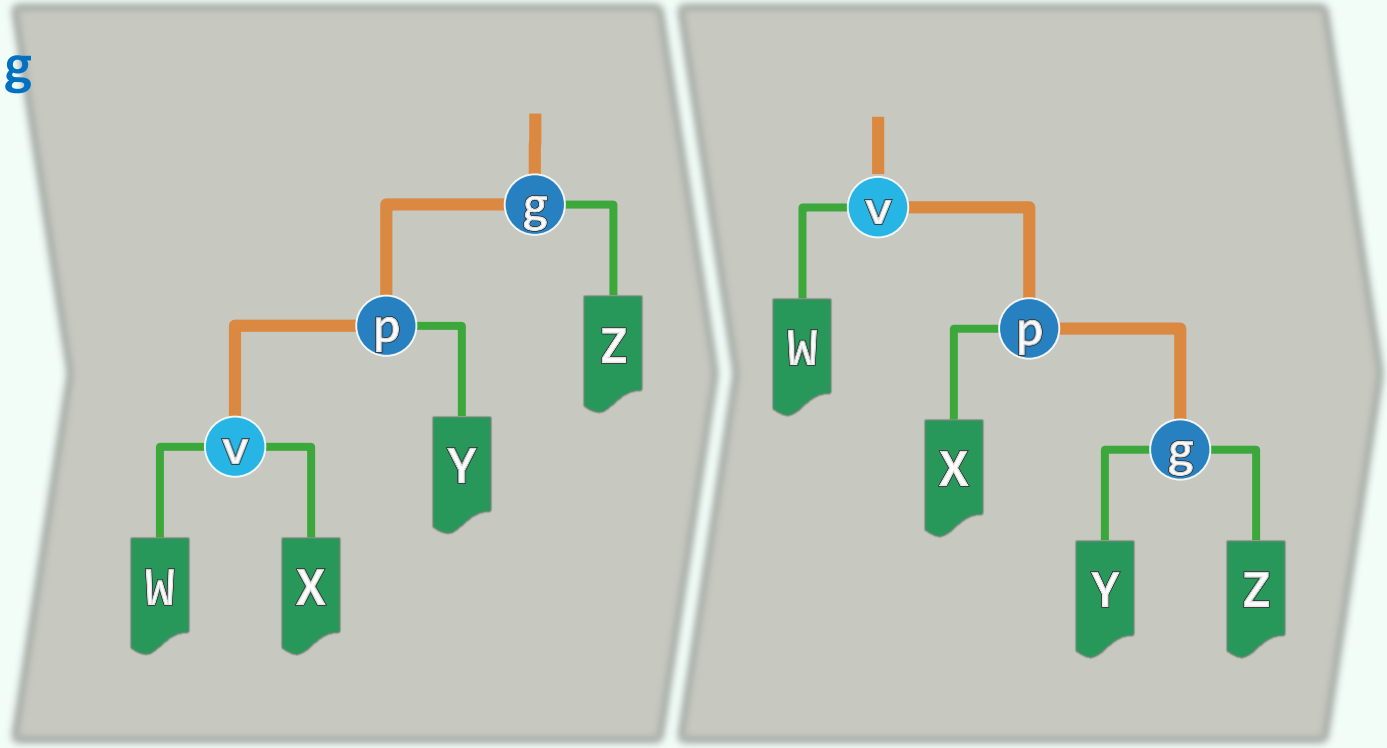
```
template <typename T> BinNodePosi<T> Splay<T>::splay( BinNodePosi<T> v ) {
   if ( ! v ) return NULL; BinNodePosi<T> p; BinNodePosi<T> g; //父亲、祖父
   while ( (p = v->parent) && (g = p->parent) ) {
      /* 自下而上，反复双层伸展 */
   }
   if ( p = v->parent ) { /* 若p果真是根，只需再额外单旋一次 */ }
   v->parent = NULL; return v; //伸展完成，v抵达树根
}
```

伸展算法：双层伸展

```
while ( (p = v->parent) && (g = p->parent) ) { //自下而上，反复双层伸展
   BinNodePosi<T> gg = g->parent; //每轮之后，v都将以原曾祖父为父
   if ( IsLChild( * v ) )
      if ( IsLChild( * p ) ) { /* zig-zig */ } else { /* zig-zag */ }
   else
      if ( IsRChild( * p ) ) { /* zag-zag */ } else { /* zag-zig */ }
   if ( !gg ) v->parent = NULL; //无曾祖父gg的v即为树根；否则，gg此后应以v为
   else ( g == gg->lc ) ? attachAsLC(v, gg) : attachAsRC(gg, v); //左或右孩子
   updateHeight( g ); updateHeight( p ); updateHeight( v );
}
```

伸展算法：举例（zig-zig）

```
if ( IsLChild( * v ) )
   if ( IsLChild( * p ) ) { //zIg-zIg
      attachAsLC( p->rc, g ); //Y
      attachAsLC( v->rc, p ); //X
      attachAsRC( p, g );
      attachAsRC( v, p );
   } else { /* zIg-zAg */ }
else
   if ( IsRChild( * p ) ) { /* zAg-zAg */ }    else { /* zAg-zIg */ }
```



查找算法

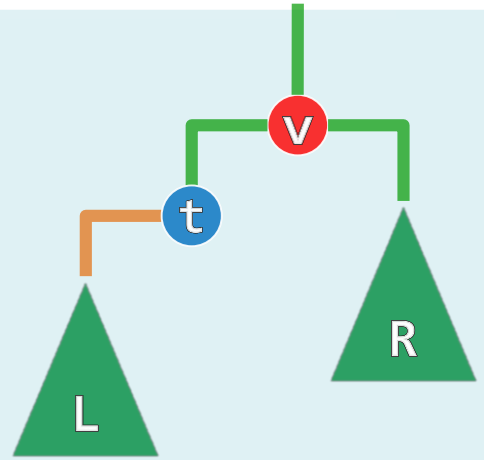
```
❖ template <typename T> BinNodePosi<T> & Splay<T>::search( const T & e ) {
   // 调用标准BST的内部接口定位目标节点
      BinNodePosi<T> p = BST<T>::search( e );
   // 无论成功与否，最后被访问的节点都将伸展至根
      _root = splay( p ? p : _hot ); //成功、失败
   // 总是返回根节点
      return _root;
   }
```

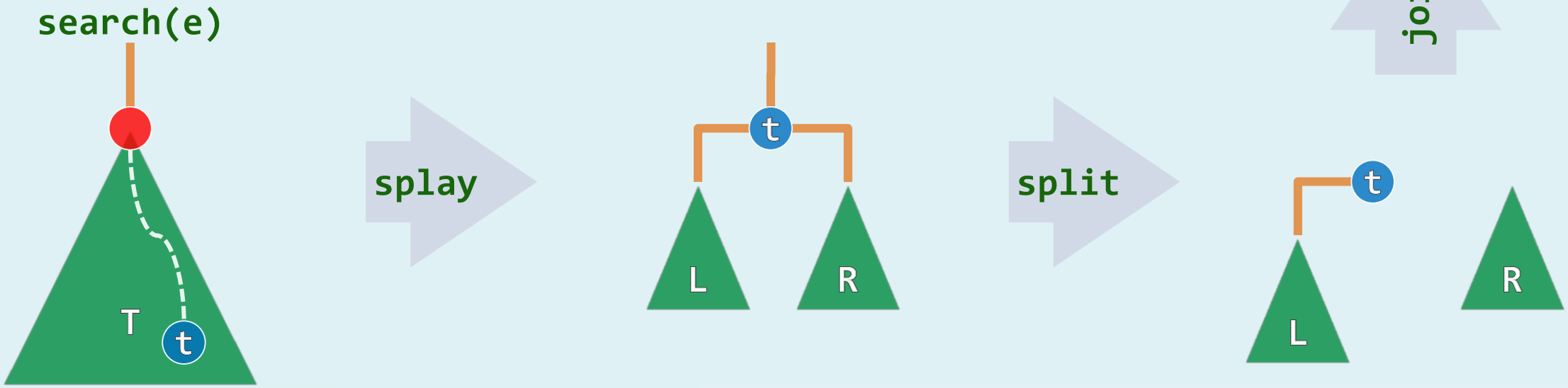
❖ 伸展树的查找，与常规BST::search()不同：很可能会改变树的拓扑结构，不再属于静态操作

插入算法

- 直观方法：先调用标准的BST::search()，再将新节点伸展至根
- 重写后的Splay::search()已集成splay()

  查找（失败）之后，_hot即是根节点
- 既如此，何不随即就在树根附近接入新节点？





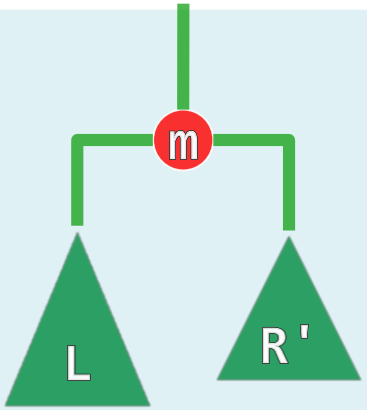
插入算法实现

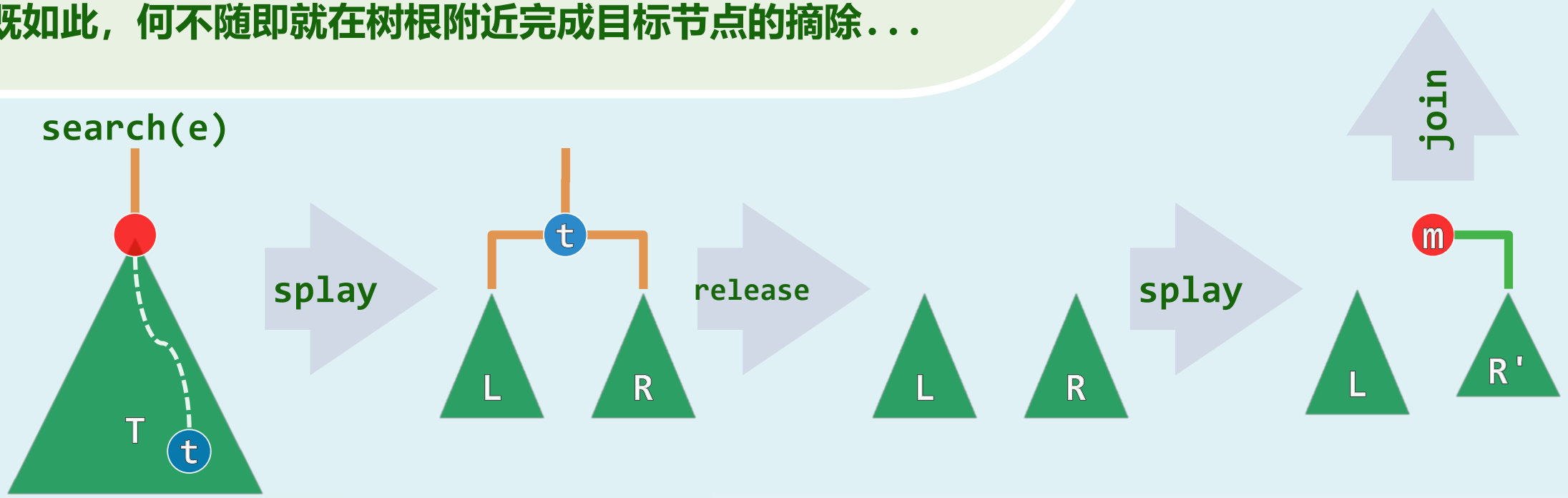
```
template <typename T> BinNodePosi<T> Splay<T>::insert( const T & e ) {
   if ( !_root ) { _size = 1; return _root = new BinNode<T>( e ); } //原树为空
   BinNodePosi<T> t = search( e ); if ( e == t->data ) return t; //t若存在，伸展至根
   if ( t->data < e ) {  //在右侧嫁接（lc == t必非空，rc或为空）
      t->parent = _root = new BinNode<T>( e, NULL, t, t->rc );
      if ( t->rc ) { t->rc->parent = _root; t->rc = NULL; }
   } else { //e < t->data，在左侧嫁接（lc或为空，rc == t必非空）
      t->parent = _root = new BinNode<T>( e, NULL, t->lc, t );
      if ( t->lc ) { t->lc->parent = _root; t->lc = NULL; }
   }
   _size++; updateHeightAbove( t ); return _root; //更新规模及t与_root的高度，插入成功
} //无论如何，返回时总有_root->data == e
```

删除算法

- 直观方法：调用BST标准的删除算法，再将_hot伸展至根
- 同样地...

  Splay::search()查找（成功）之后，目标节点即是树根
- 既如此，何不随即就在树根附近完成目标节点的摘除...





删除算法实现

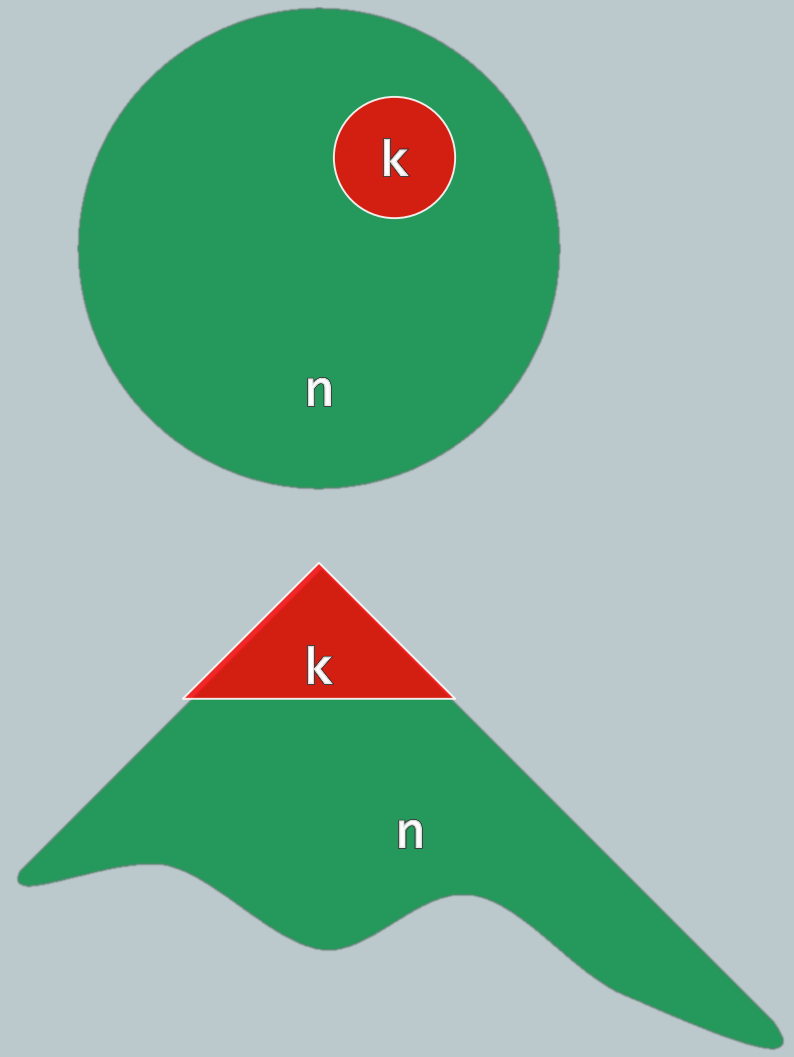
```
template <typename T> bool Splay<T>::remove( const T & e ) {
   if ( !_root || ( e != search( e )->data ) ) return false; //若目标存在，则伸展至根
   BinNodePosi<T> L = _root->lc, R = _root->rc; release(t); //记下左、右子树后，释放之
   if ( !R ) { 若R空
      if ( L ) L->parent = NULL; _root = L; //则L即是余树
   } else { //否则
      _root = R; R->parent = NULL; search( e ); //在R中再找e：注定失败，但最小节点必
      if (L) L->parent = _root; _root->lc = L; //伸展至根，故可令其以L作为左子树
   }
    _size--; if ( _root ) updateHeight( _root ); //更新记录
   return true; //删除成功
}
```

综合评价

- 无需记录高度或平衡因子；编程实现简单——优于AVL树

  分摊复杂度 $\mathcal{O}(\log n)$ ——与AVL树相当

- 局部性强、缓存命中率极高时（即 $k \ll n \ll m$ ）
  - 效率甚至可以更高——自适应的 $\mathcal{O}(\log k)$
  - 任何连续的m次查找，仅需 $\mathcal{O}(m \log k + n \log n)$ 时间
- 若反复地顺序访问任一子集，分摊成本仅为常数
- 不能杜绝单次最坏情况，不适用于对效率敏感的场合
- 复杂度的分析稍嫌复杂——好在有初等的证明...



高级搜索树
伸展树：分摊分析

S的势能

❖ （任何时刻的）任何一棵伸展树S，都可以假想地被认为具有势能：

$$\Phi(S) = \log\left(\prod_{v\in S} size(v)\right) = \sum_{v\in S}\log\left(size(v)\right) = \sum_{v\in S} rank(v)$$

❖ 直觉：越平衡/倾侧的树，势能越小/大

- 单链：$\Phi(S) = \log n! = \mathcal{O}(n\log n)$

- 满树：$\Phi(S) = \log\prod_{d=0}^{h}(2^{h-d+1}-1)^{2^d} \le \log\prod_{d=0}^{h}(2^{h-d+1})^{2^d}$

$$= \log\prod_{d=0}^{h} 2^{(h-d+1)\cdot 2^d} = \sum_{d=0}^{h}(h-d+1)\cdot 2^d = (h+1)\cdot\sum_{d=0}^{h}2^d - \sum_{d=0}^{h} d\cdot 2^d$$

$$= (h+1)\cdot(2^{h+1}-1) - [(h-1)\cdot 2^{h+1} + 2] = 2^{h+2} - h - 3 = \mathcal{O}(n)$$

T的上界

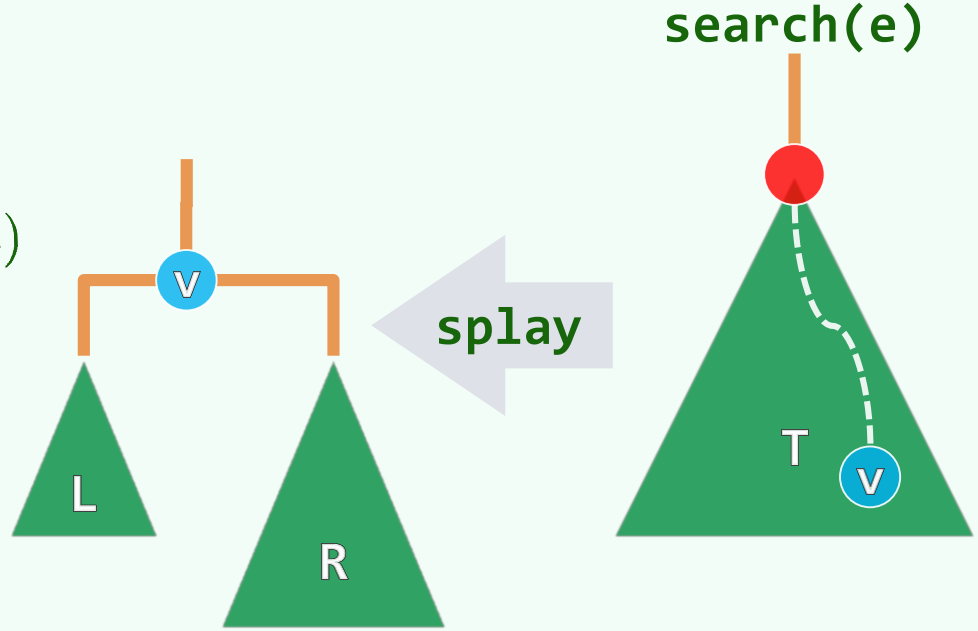
❖ 考查对伸展树S 的 $m \gg n$ 次连续访问（不妨仅考查search()）

❖ 若记：$A^{(k)} = T^{(k)} + \Delta\Phi^{(k)},\ k = 0, 1, 2, \ldots, m$

则有：$A - \mathcal{O}(n \log n) \leq \underline{T = A - \Delta\Phi} \leq A + \mathcal{O}(n \log n)$

❖ 故若能证明：$A = \mathcal{O}(m \log n)$

则必有：$T = \mathcal{O}(m \log n)$

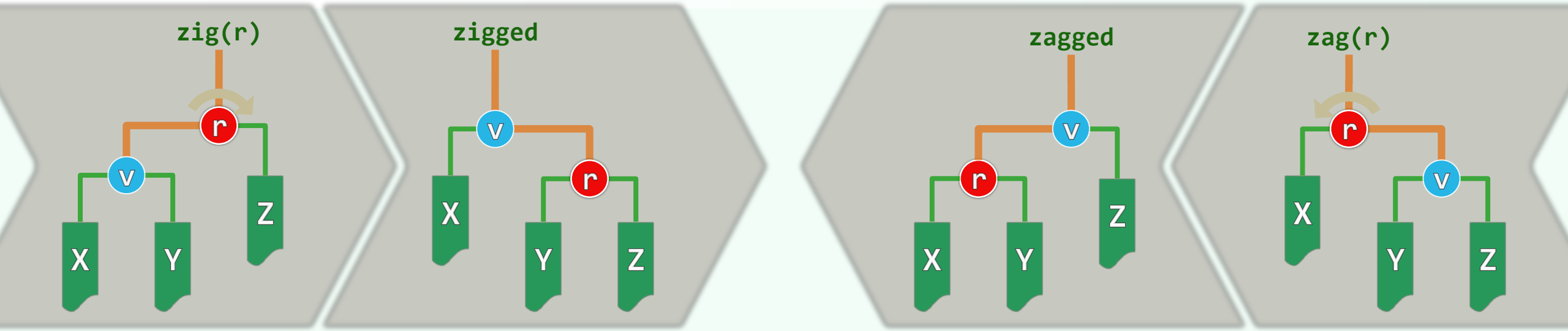


❖ 好消息是，尽管$T^{(k)}$ 的变化幅度可能很大，我们却能证明：

$A^{(k)}$ 都不致超过节点v的势能变化量，即：$\mathcal{O}(\ rank^{(k)}(v) - rank^{(k-1)}(v)\ ) = \mathcal{O}(\log n)$

❖ 事实上，$A^{(k)}$ 不过是v的若干次连续伸展操作（时间成本）的累积，这些操作无非三种情况...

Zig / Zag

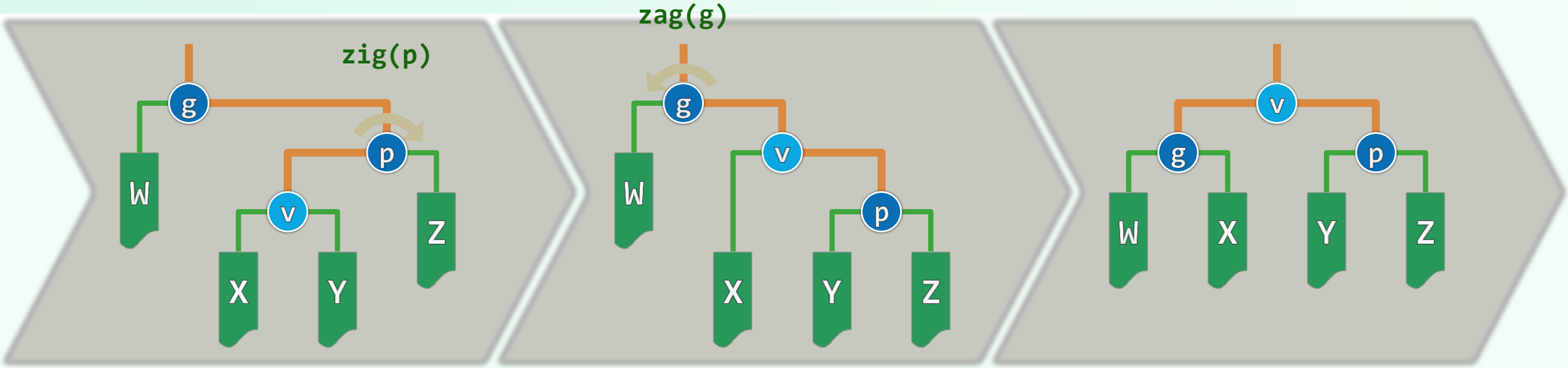


$$A_i^{(k)} = T_i^{(k)} + \Delta\Phi(S_i^{(k)}) = 1 + \Delta rank_i(v) + \Delta rank_i(r)$$

$$= 1 + [rank_i(v) - rank_{i-1}(v)] + [\underline{rank_i(r) - rank_{i-1}(r)}]$$

$$< 1 + [rank_i(v) - rank_{i-1}(v)]$$

zig-zag / zag-zig



$$A_i^{(k)} = T_i^{(k)} + \Delta\Phi(S_i^{(k)}) = 2 + \Delta rank_i(g) + \Delta rank_i(p) + \Delta rank_i(v)$$

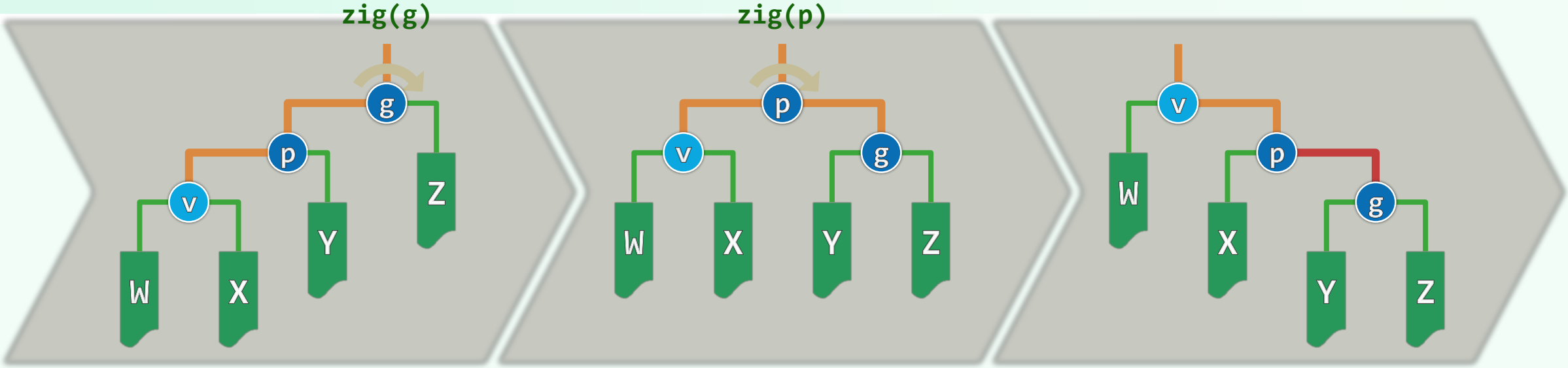
$$= 2 + [rank_i(g) - \cancel{rank_{i-1}(g)}] + [rank_i(p) - \underline{rank_{i-1}(p)}] + [\cancel{rank_i(v)} - \underline{rank_{i-1}(v)}]$$

$$< 2 + \underline{rank_i(g) + rank_i(p)} - 2 \cdot rank_{i-1}(v) \qquad (\because\ rank_{i-1}(p) > rank_{i-1}(v))$$

$$< \cancel{2} + \underline{2 \cdot rank_i(v) - \cancel{2}} - 2 \cdot rank_{i-1}(v) \qquad (\because\ \frac{\log G_i + \log P_i}{2} \le \log \frac{G_i + P_i}{2} < \log \frac{V_i}{2})$$

$$= 2 \cdot (rank_i(v) - rank_{i-1}(v))$$

zig-zig / zag-zag



$$A_i^{(k)} = T_i^{(k)} + \Delta\Phi(S_i^{(k)}) = 2 + \Delta rank_i(g) + \Delta rank_i(p) + \Delta rank_i(v)$$

$$= 2 + [rank_i(g) - \cancel{rank_{i-1}(g)}] + [rank_i(p) - \underline{rank_{i-1}(p)}] + [\cancel{rank_i(v)} - \underline{rank_{i-1}(v)}]$$

$$< 2 + rank_i(g) + \underline{rank_i(p)} - 2 \cdot rank_{i-1}(v) \qquad (\because\ rank_{i-1}(p) > rank_{i-1}(v))$$

$$< 2 + rank_i(g) + \underline{rank_i(v)} - 2 \cdot rank_{i-1}(v) \qquad (\because\ rank_i(p) < rank_i(v))$$

$$< 3 \cdot (rank_i(v) - rank_{i-1}(v)) \qquad (\because\ \frac{\log G_i + \log V_{i-1}}{2} \le \log \frac{G_i + V_{i-1}}{2} < \log \frac{V_i}{2})$$