Splay Trees

Zhengwei Qi

Part I: Basic Concepts

8.高级搜索树

（a1）伸展树：逐层伸展

我要一步一步往上爬

在最高点乘着叶片往前飞

邓俊辉

deng@tsinghua.edu.cn

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 //BST就是这样的一个例子

- BST：刚刚被访问过的节点，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

- 连续的m次查找（m >> n = |BST|），采用AVL共需O(mlogn)时间

- 利用局部性，能否更快? //仿效自适应链表

- 策略：节点一旦被访问，随即调整至树根 //如此，下次访问即可...

- 问题：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

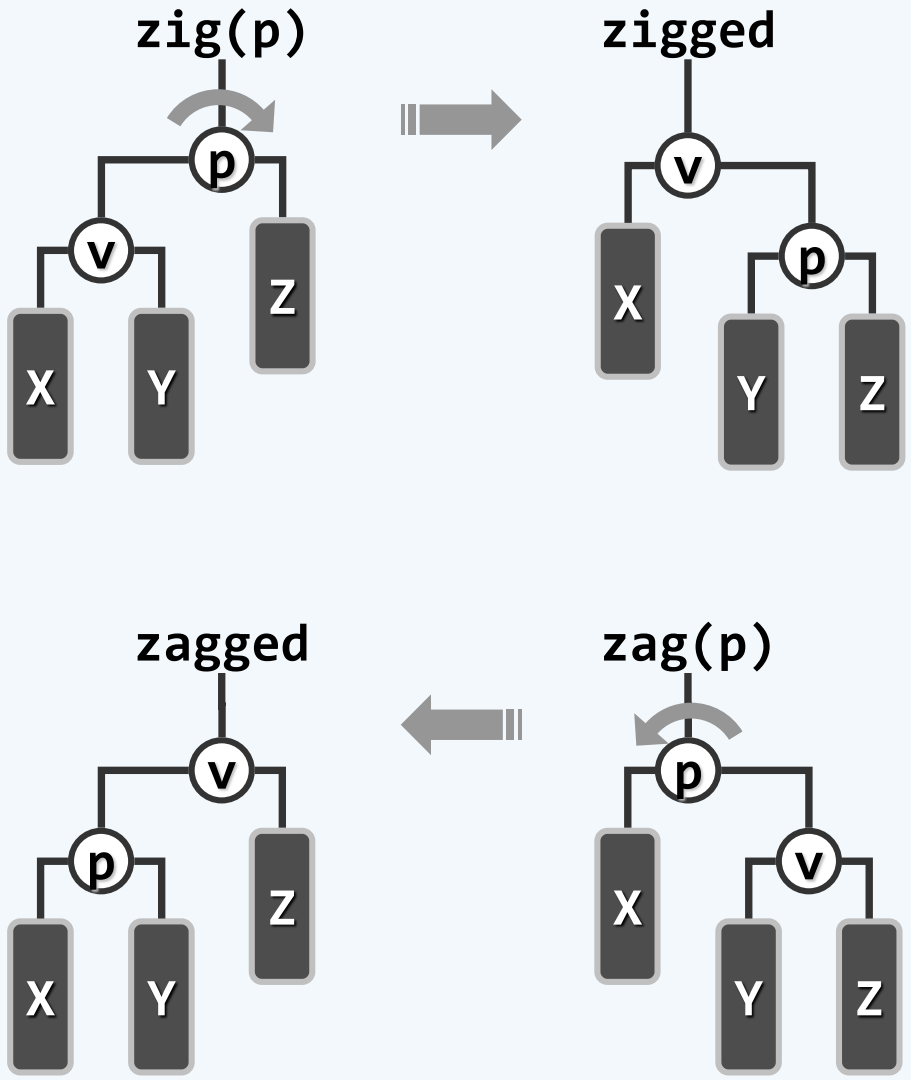
- 节点v一旦被访问，随即转移至树根
- 一步一步往上爬

自下而上，逐层单旋

```
zig( v->parent )

zag( v->parent )
```

直到v最终被推送至根



实例

- 伸展过程的效率

  是否足够地高?

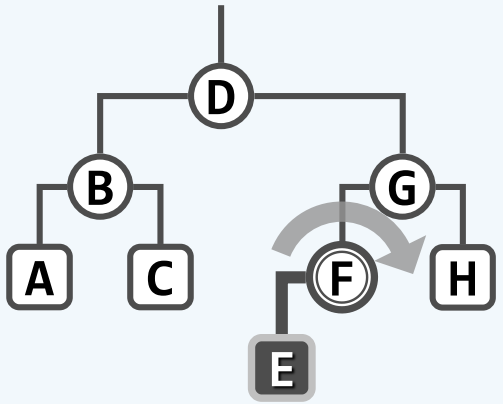
- 就逐层伸展的策略而言

  这取决于

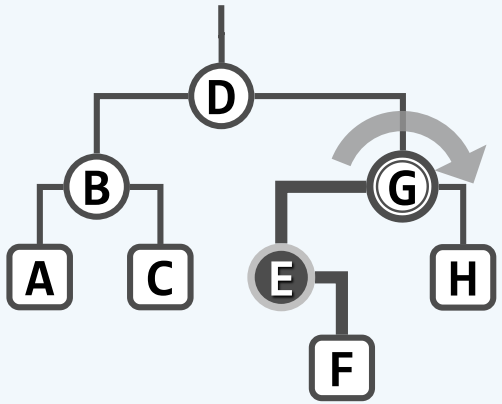
  树的初始形态和

  节点的访问次序

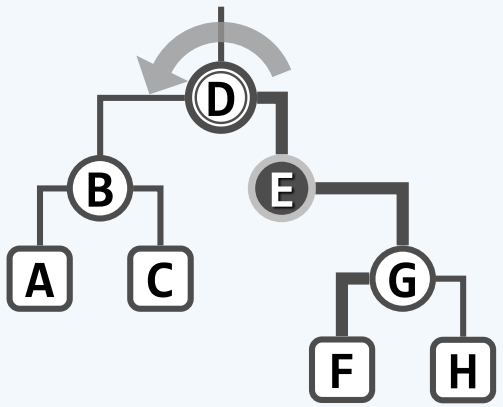
访问E之后，做zig(F)
(a)



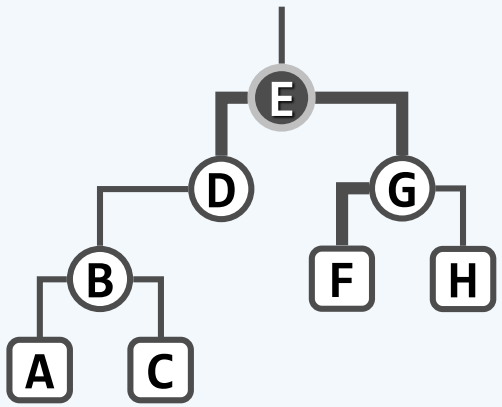
zig(G)
(b)



zag(D)
(c)

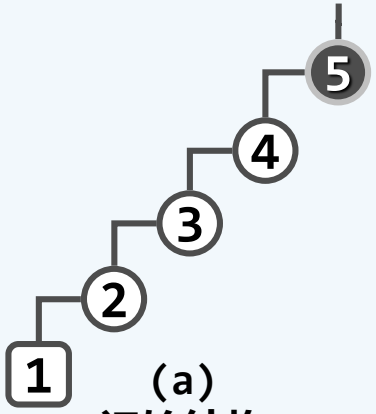


等价变换：经3次旋转，E调整至树根
(d)

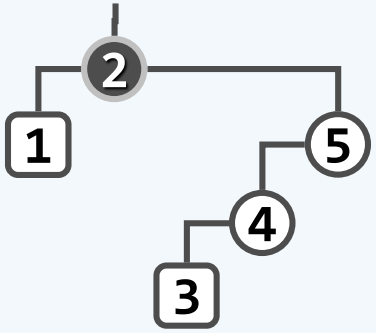


最坏情况

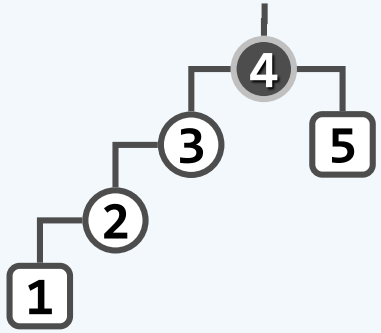
❖ 旋转次数呈周期性的算术级数演变：每一周期累计$\Omega(n^2)$，分摊$\Omega(n)$！



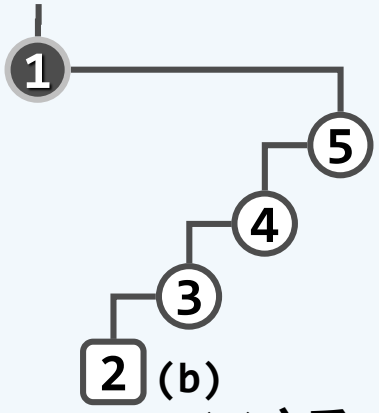
(a)
初始结构



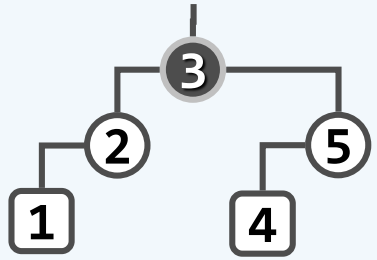
(c)
search(2)之后



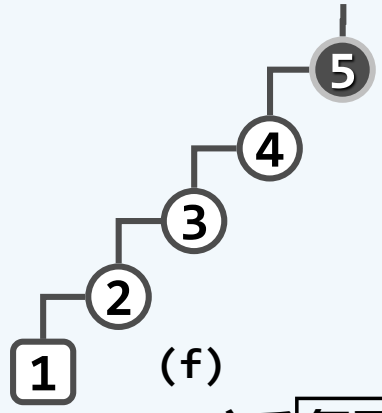
(e)
search(4)之后



(b)
search(1)之后



(d)
search(3)之后



(f)
search(5)之后复原

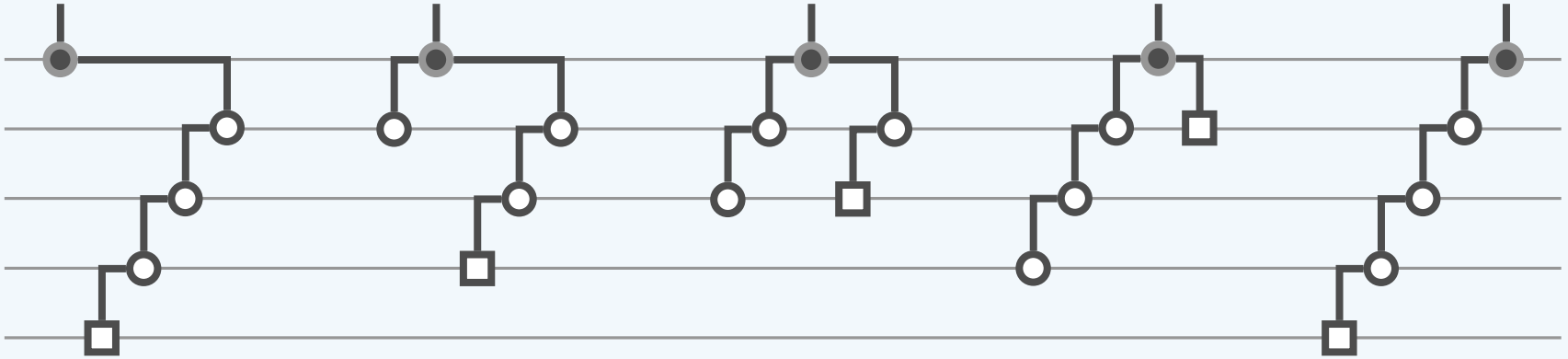
低效率的根源

❖ 最坏情况，问题出在哪里?

1)全树拓扑始终呈单链条结构，等价于一维列表

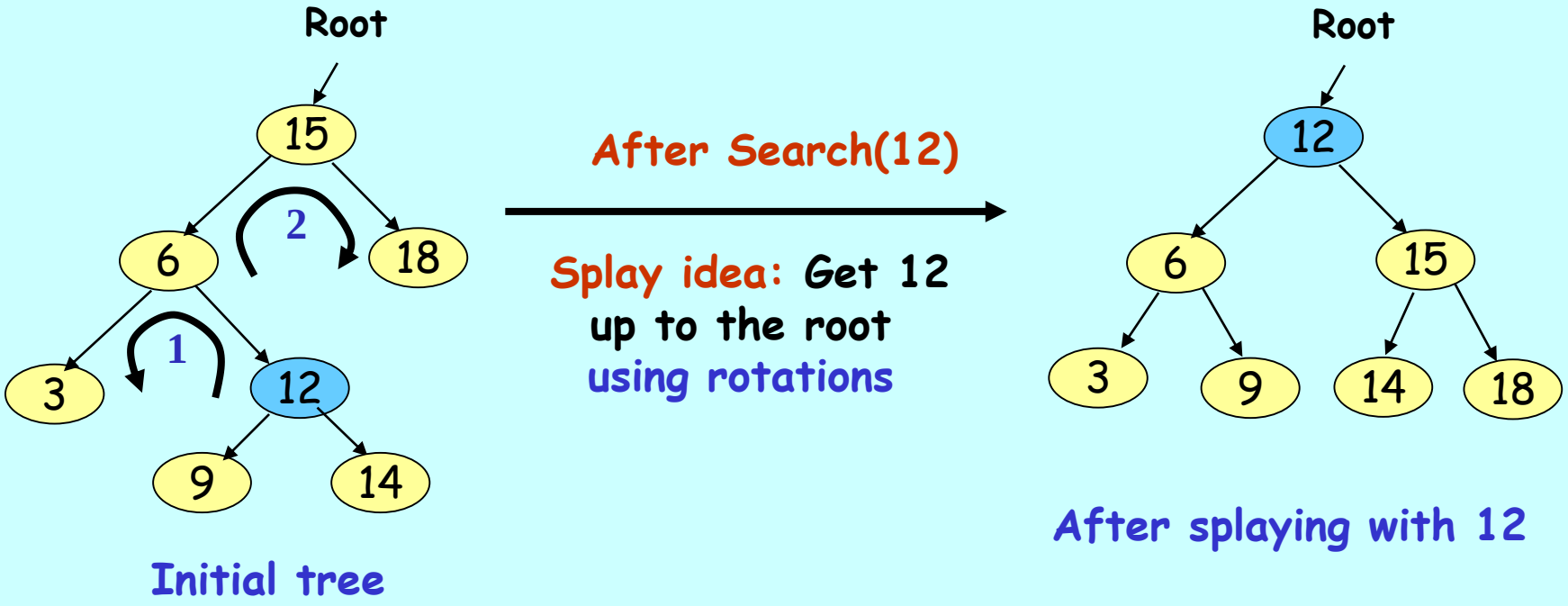
2)被访问节点的深度，呈周期性的算术级数演变，平均为Ω(n)：

n - 1, n - 2, n - 3, ..., 3, 2, 1; n – 1, ...



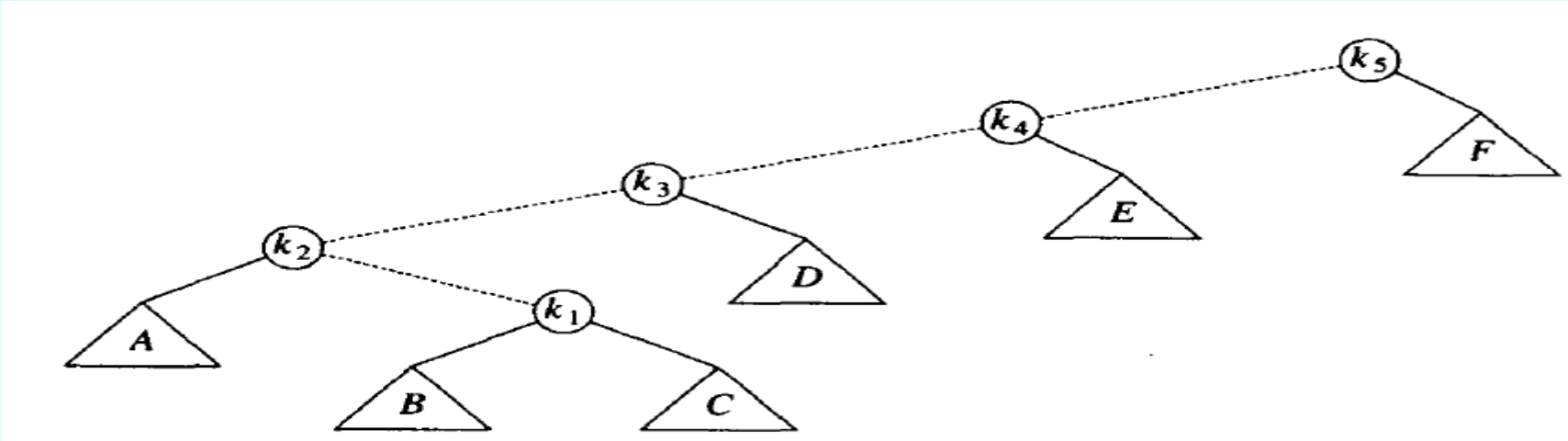
❖ 问题的症结既已确定，便可针对性地改进...

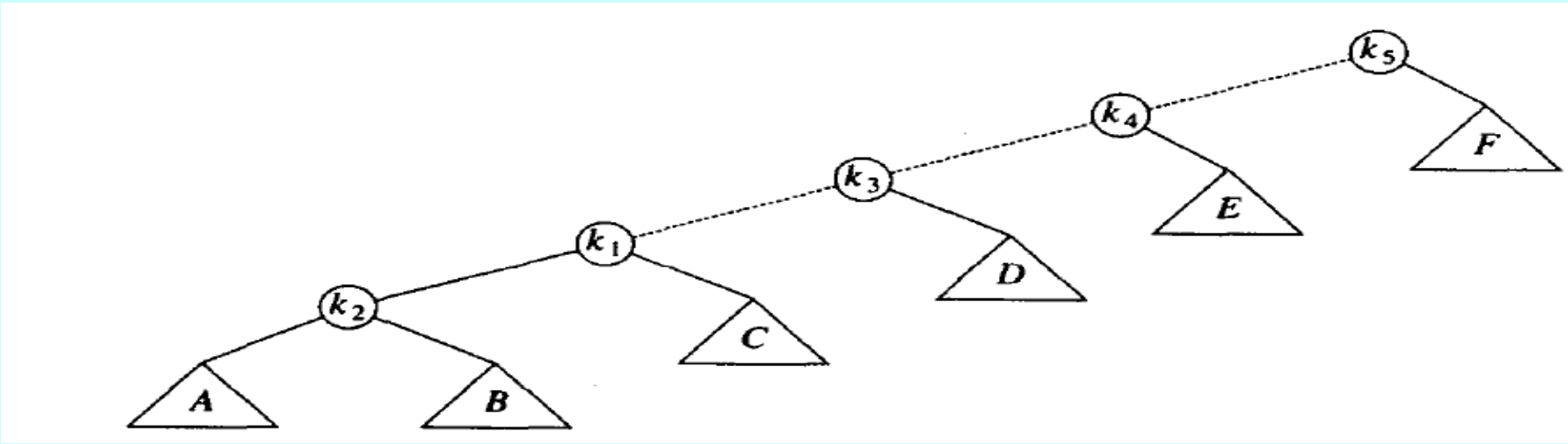
Example



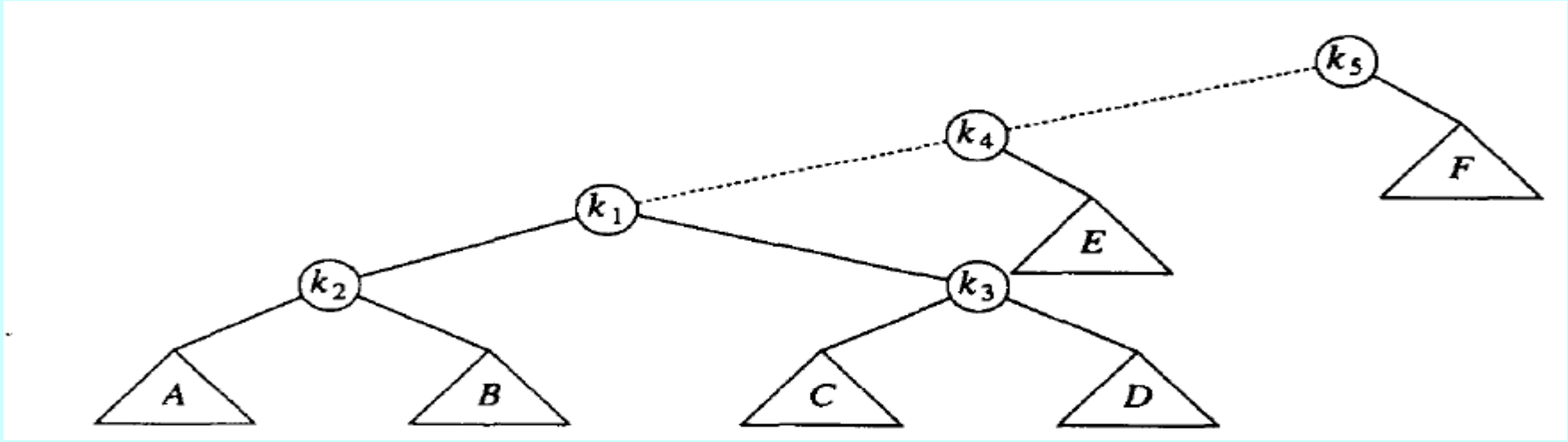
- Not only splaying with 12 makes the tree balanced, subsequent accesses for 12 will take O(1) time.
- Active (recently accessed) nodes will move towards the root and inactive nodes will slowly move further from the root

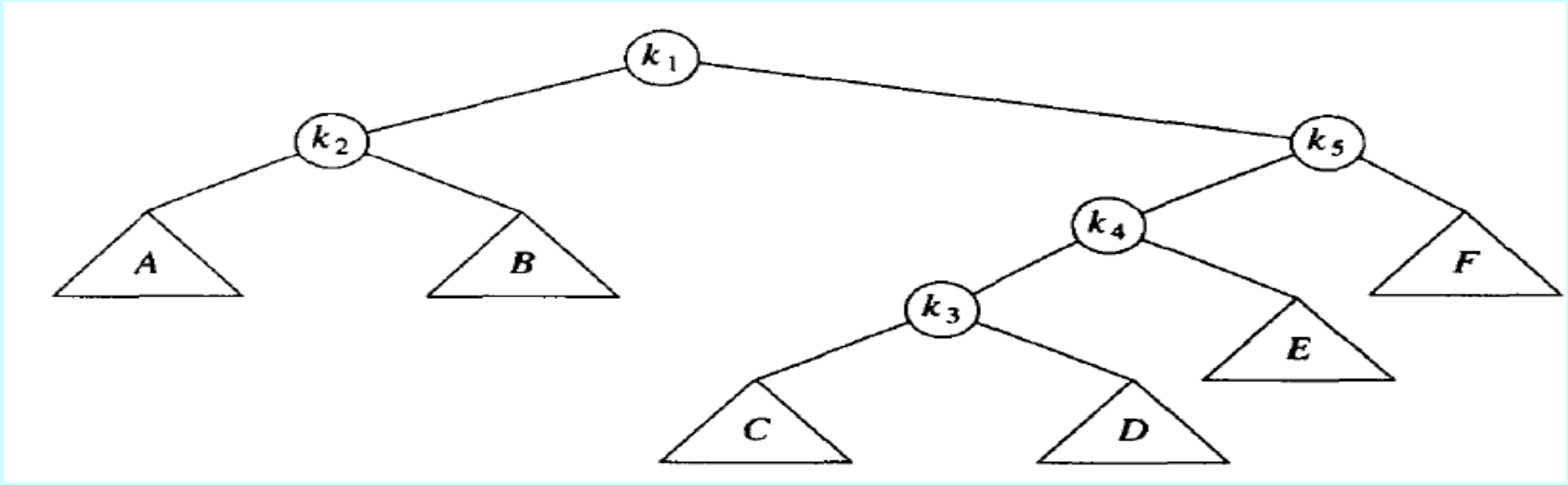
Naïve Solutions





Naïve Solutions-cont.





8.高级搜索树

（a2）伸展树：双层伸展

贾政道：“不用全打开，怕叠起来倒费事。”
詹光便与冯紫英一层一层折好收拾。

邓俊辉

deng@tsinghua.edu.cn

双层伸展

❖ D. D. Sleator & R. E. Tarjan

Self-Adjusting Binary Trees

J. ACM, 32:652-686, 1985

❖ 构思的精髓：向上追溯两层，而非一层

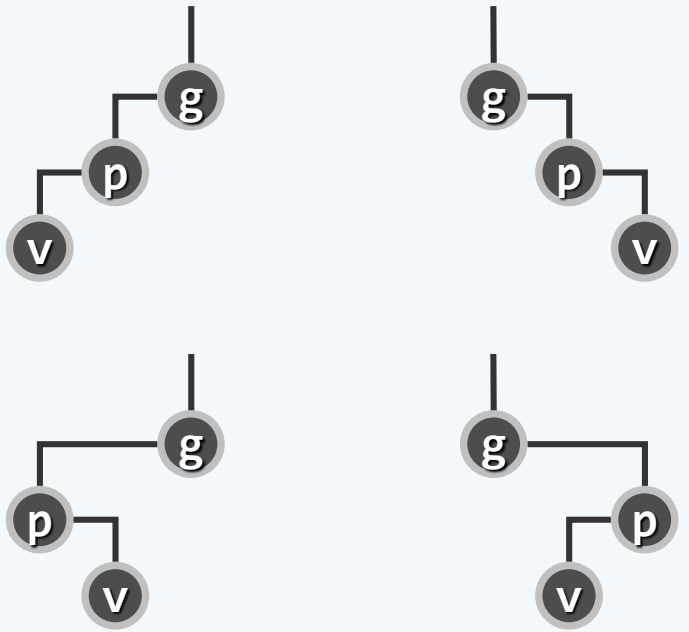
❖ 反复考察祖孙三代：g = parent(p), p = parent(v), v

❖ 根据它们的相对位置，经两次旋转使得

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?



Self-adjusting binary search trees

Self-Adjusting Binary Search Trees

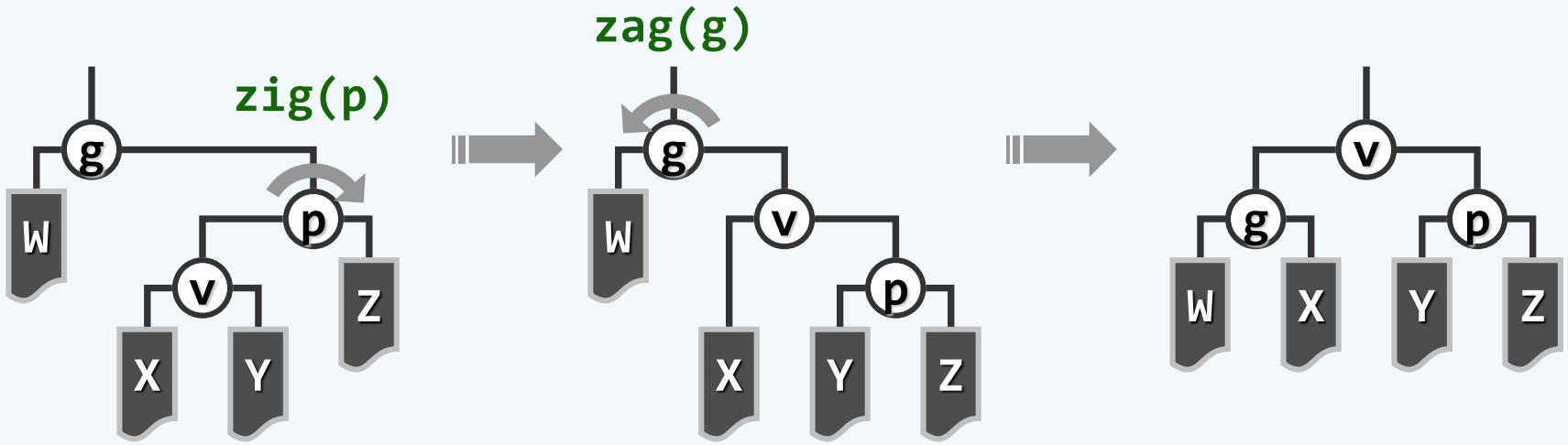
DANIEL DOMINIC SLEATOR AND ROBERT ENDRE TARJAN

AT&T Bell Laboratories, Murray Hill, NJ

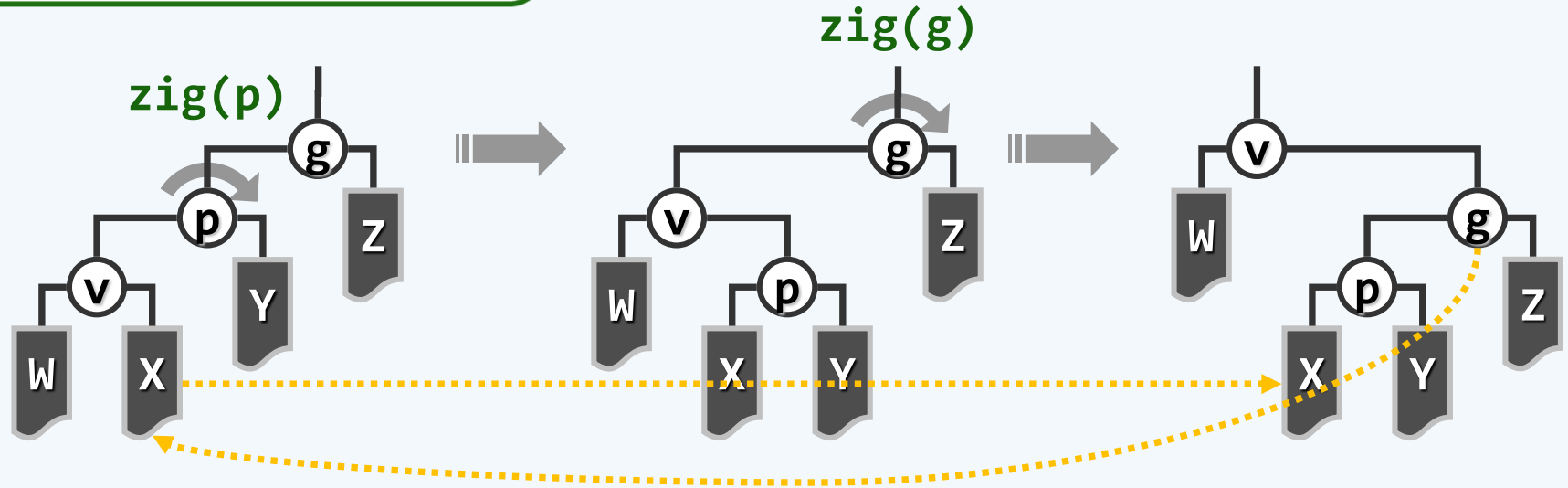
Abstract. The *splay* tree, a self-adjusting form of binary search tree, is developed and analyzed. The binary search tree is a data structure for representing tables and lists so that accessing, inserting, and deleting items is easy. On an n-node splay tree, all the standard search tree operations have an amortized time bound of $O(\log n)$ per operation, where by "amortized time" is meant the time per operation averaged over a worst-case sequence of operations. Thus splay trees are as efficient as balanced trees when total running time is the measure of interest. In addition, for sufficiently long access sequences, splay trees are as efficient, to within a constant factor, as static optimum search trees. The efficiency of splay trees comes not from an explicit structural constraint, as with balanced trees, but from applying a simple restructuring heuristic, called *splaying*, whenever the tree is accessed. Extensions of splaying give simplified forms of two other data structures: lexicographic or multidimensional search trees and link/cut trees.

zig-zag / zag-zig

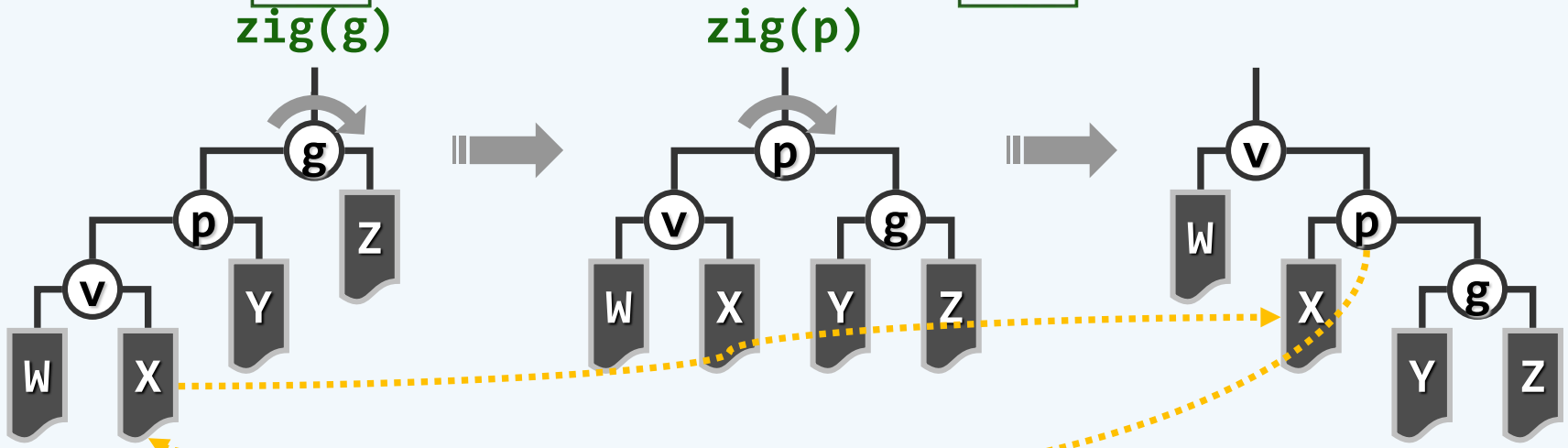
- 与AVL树双旋完全等效！
- 与逐层伸展别无二致！
- 难道，就这样平淡无奇？



zig-zig / zag-zag



❖ 颠倒次序之后，局部的细微差异，将彻底地改变整体 ...

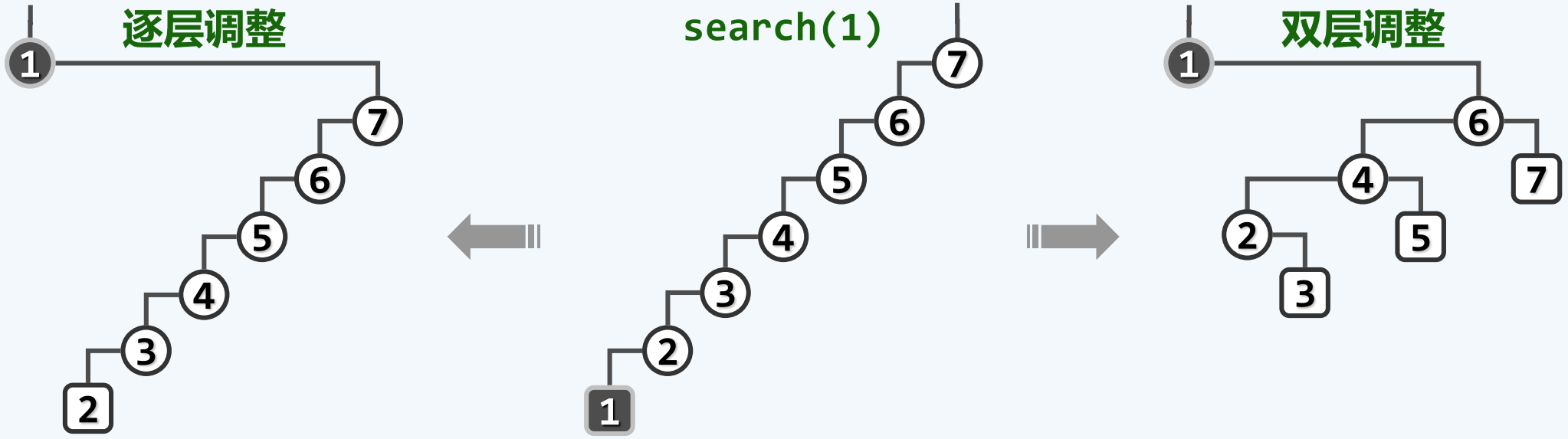


zig-zig / zag-zag

❖ 折叠效果：一旦访问坏节点，对应路径的长度将随即减半　　//含羞草

❖ 最坏情况不致持续发生！

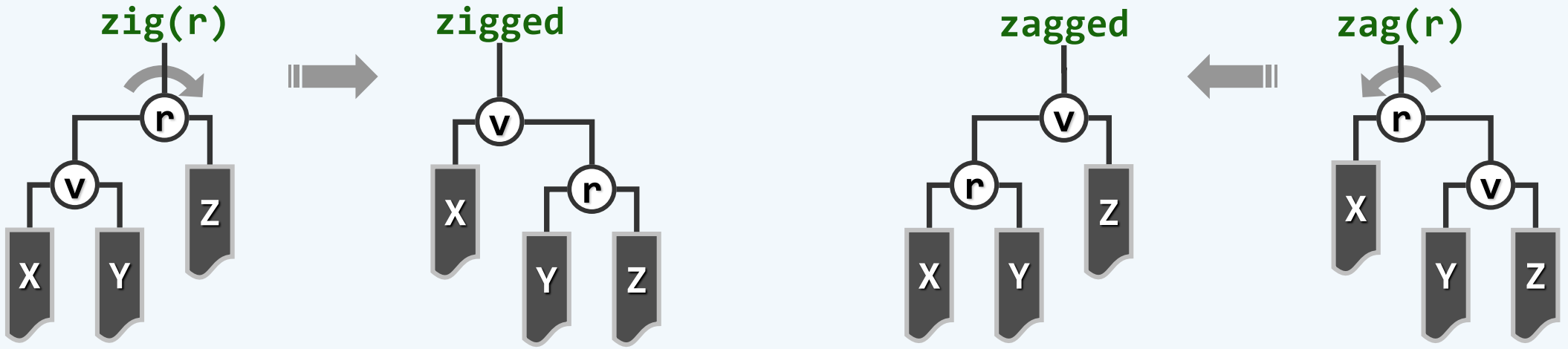
单趟伸展操作，分摊$O(\log n)$时间！　　//严格证明，详见习题[8-2]



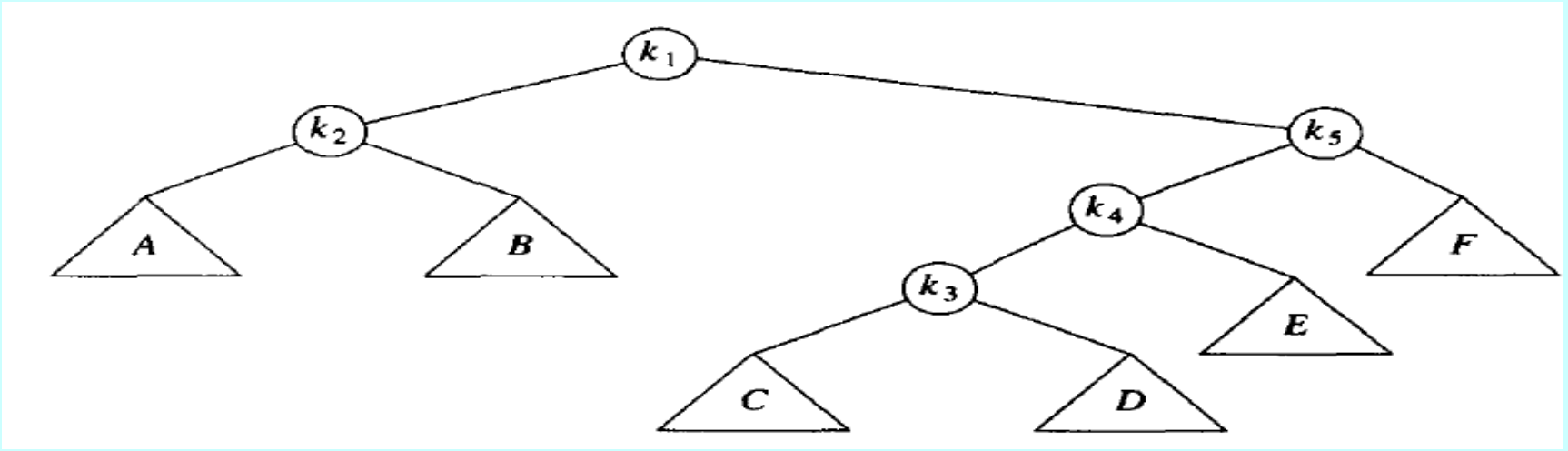
zig / zag

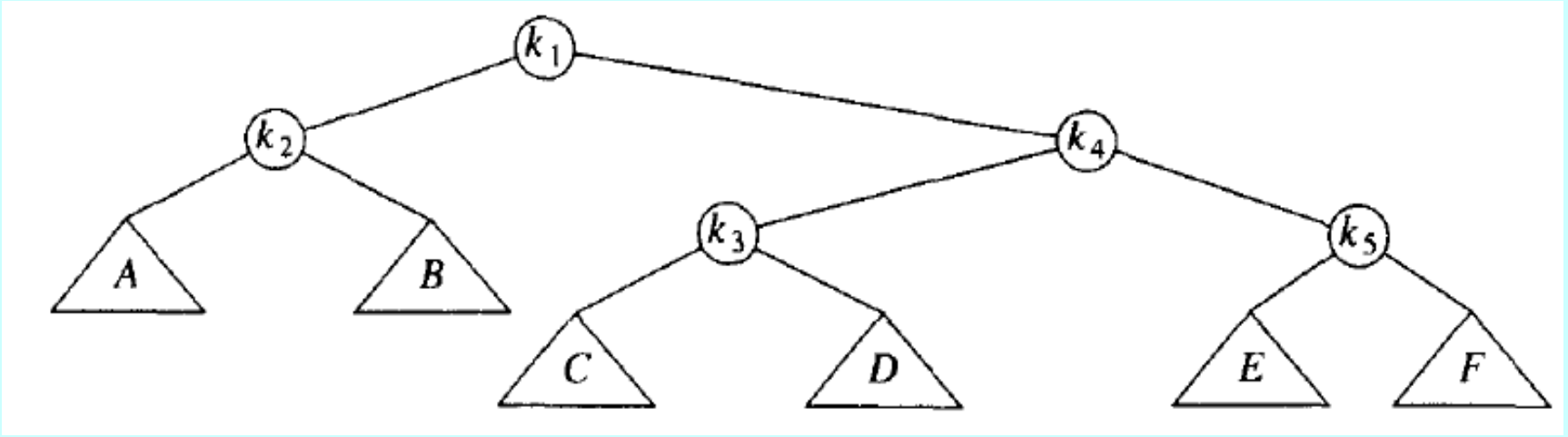
- 要是v只有父亲，没有祖父呢?
- 此时必有`parent(v) == root(T)`，且

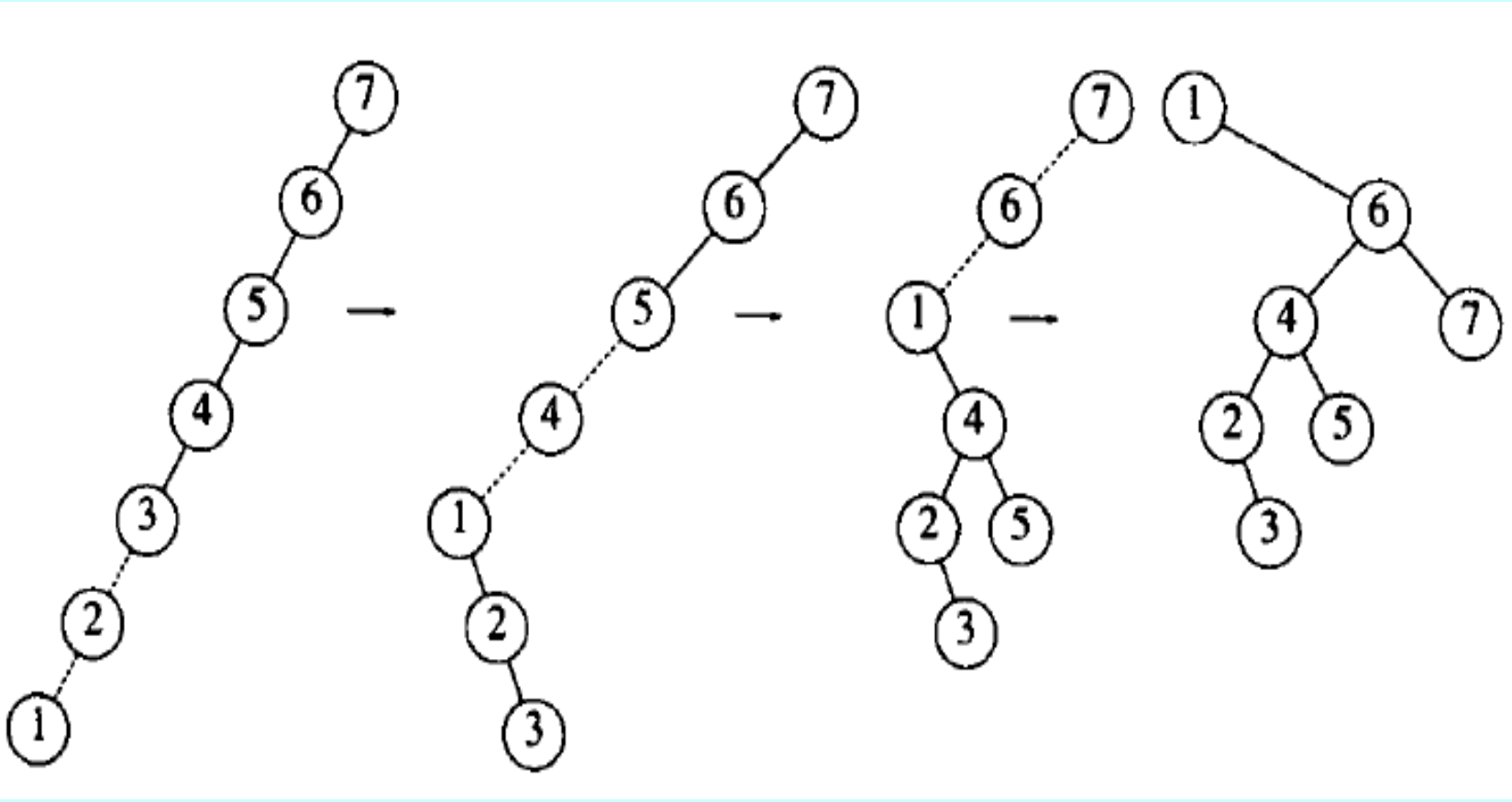
  每轮调整中，这种情况至多（在最后）出现一次
- 视具体形态，做单次旋转：zig(r)或zag(r)



Naïve vs Bottom Up






7
6
5
4
3
2
1
7
6
5
4
1
2
3
7
6
1
4
2
5
3
1
6
4
7
2
5
3

8.高级搜索树

（a3）伸展树：算法实现

到了所在，住了脚，便把这驴似纸一般折叠起来，其厚也只比张纸，放在巾箱里面。

邓俊辉

deng@tsinghua.edu.cn

伸展树接口

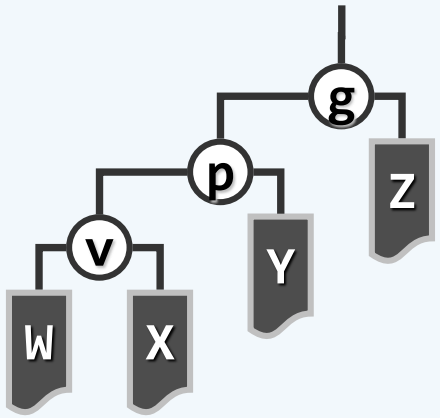
```
template <typename T>
class Splay : public BST<T> { //由BST派生
protected: BinNodePosi(T) splay( BinNodePosi(T) v ); //将v伸展至根
public: //伸展树的查找也会引起整树的结构调整，故search()也需重写
   BinNodePosi(T) & search( const T & e ); //查找 重写
   BinNodePosi(T) insert( const T & e ); //插入 重写
   bool remove( const T & e ); //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

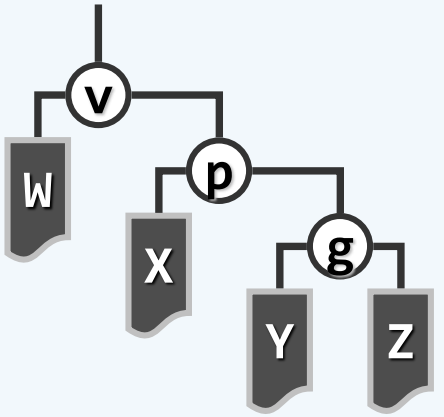
伸展算法

```
❖ 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
      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; //若无曾祖父gg，则v现即为树根；否则，gg此后应以v为左或右
      else ( g == gg->lc ) ? attachAsLChild(gg, v) : attachAsRChild(gg, v); //孩子
      updateHeight( g ); updateHeight( p ); updateHeight( v );
    } //双层伸展结束时，必有g == NULL，但p可能非空
    if ( p = v->parent ) { /* 若p果真是根，只需在额外单旋（至多一次） */ }
    v->parent = NULL; return v; //伸展完成，v抵达树根
  }
```

伸展算法

```
❖if ( IsLChild( * v ) )
    if ( IsLChild( * p ) ) { //zIg-zIg
       attachAsLChild( g, p->rc );
       attachAsLChild( p, v->rc );
       attachAsRChild( p, g );
       attachAsRChild( 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 = searchIn( _root, e, _hot = NULL );
// 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;
}
```

- 伸展树的查找操作，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

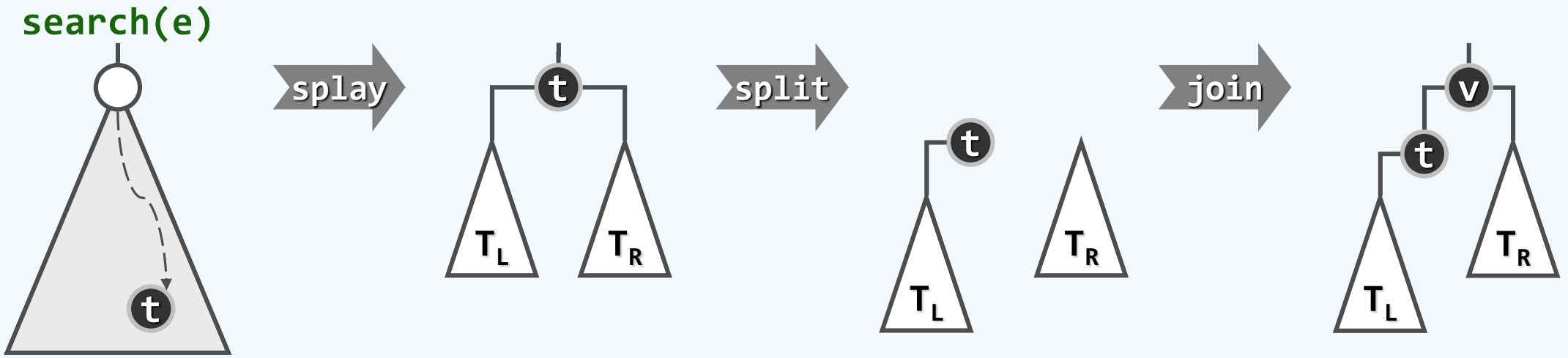
- 直观方法：调用BST标准的插入算法，再将新节点伸展至根

  其中，首先需调用BST::search()

- 重写后的Splay::search()已集成了splay()操作

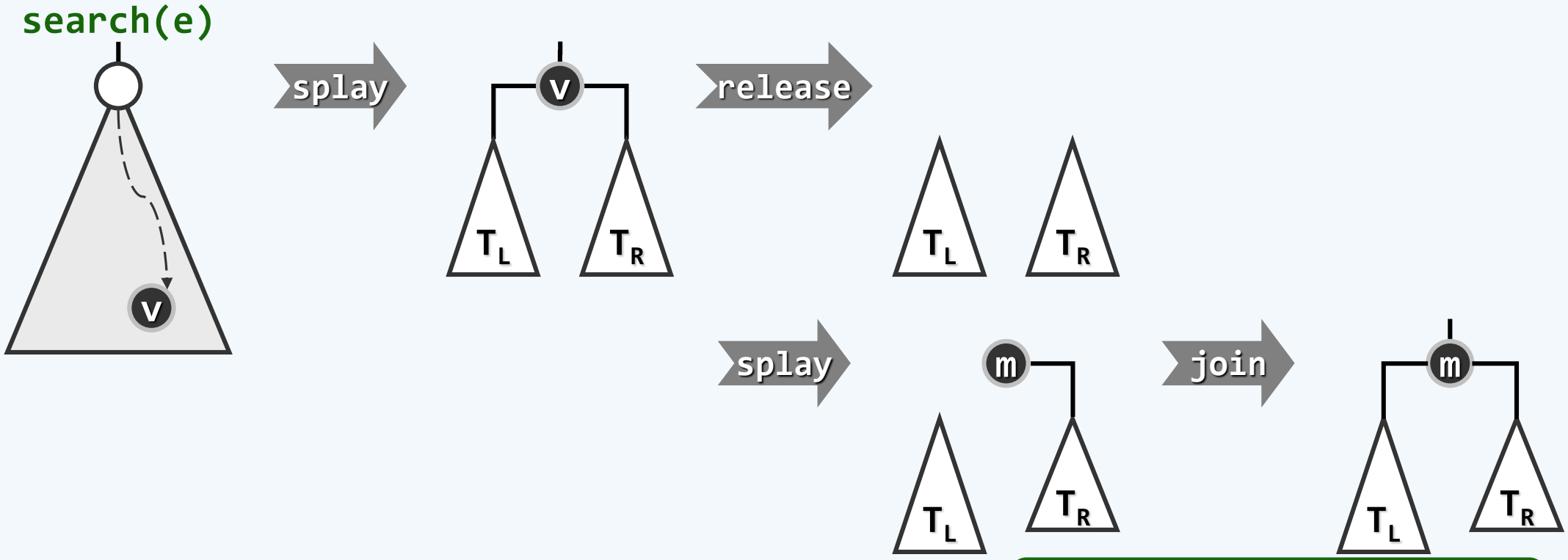
  查找（失败）之后，_hot即是根节点

- 既如此，何不随即就在树根附近完成新节点的接入...



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...



综合评价

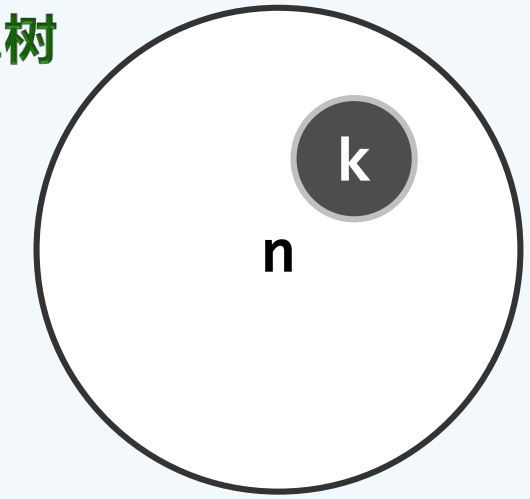
- 无需记录节点高度或平衡因子；编程实现简单易行——优于AVL树

  分摊复杂度$O(\log n)$——与AVL树相当

- 局部性强、缓存命中率极高时（即$k \ll n \ll m$）

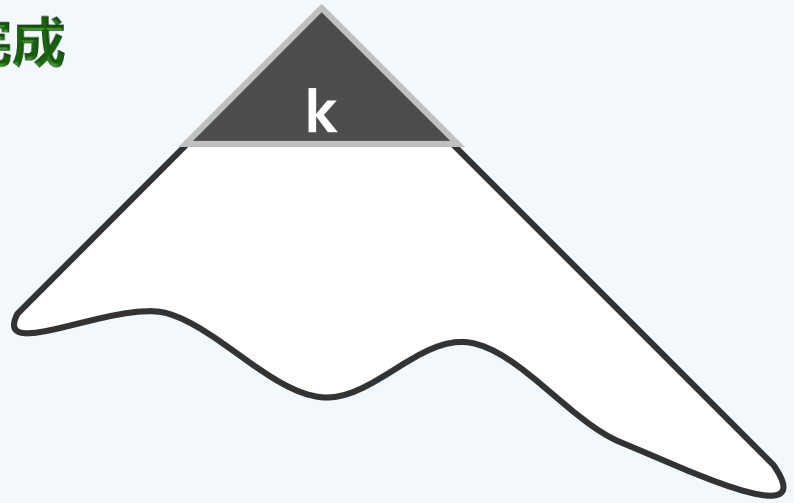
  效率甚至可以更高——自适应的$O(\log k)$

  任何连续的m次查找，都可在$O(m \log k + n \log n)$时间内完成



- 仍不能杜绝单次最坏情况的出现

  不适用于对效率敏感的场合



- 复杂度的分析稍嫌复杂——好在有初等的方法

Part II: Top-down splaying

http://users.cis.fiu.edu/~weiss/dsaa_c++3/code/SplayTree.h

Splay trees: top-down splaying

- In bottom-up splaying we traveled the tree down to locate the node to be splayed and then up to perform the splaying.
- In top-down splaying
  - We travel the tree only once (down).
  - As we descend, we take the nodes that are found on the access path and move them out of the way while restructuring the “pieces”
  - In the end we put everything back together.
- Time
  - The amortized time is the same (logarithmic) but in practice this method is faster.

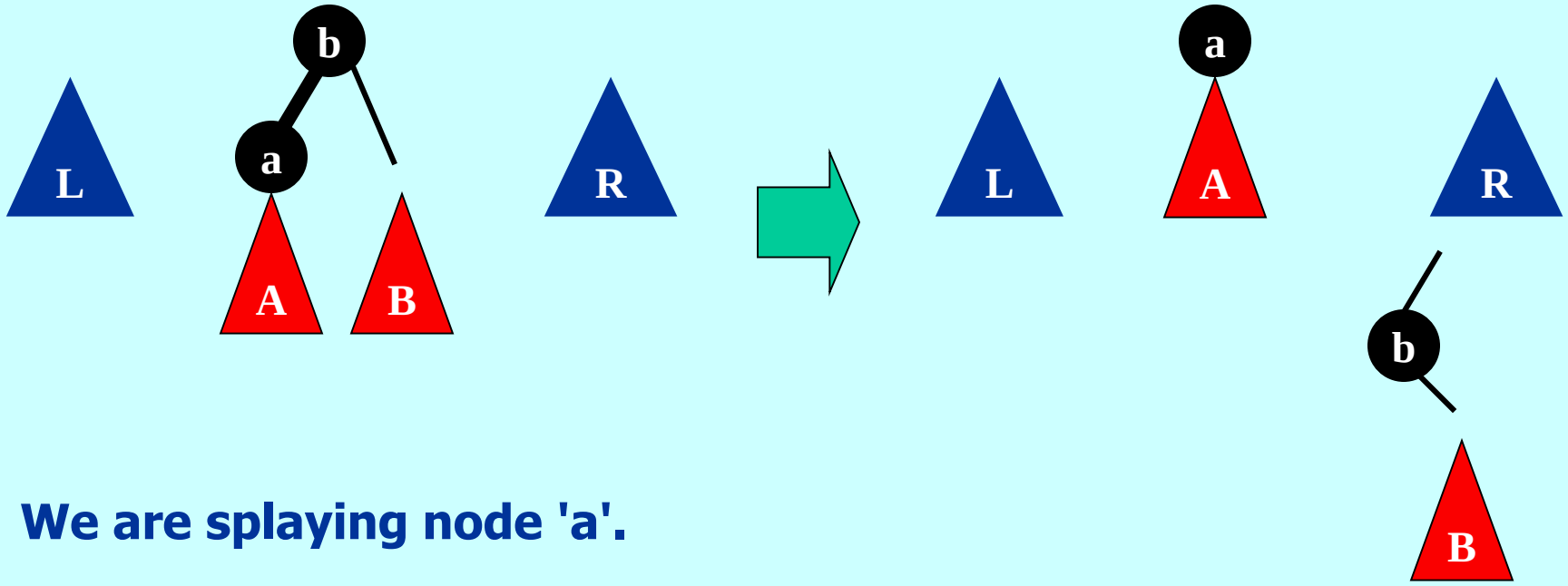
Splay trees: top-down splaying

- At any time during the splaying of a node x, we have three subtrees:
  - L = subtree containing nodes smaller than x, encountered on the access path towards x.
  - T = subtree containing x
    - Its root is always the current node on the search.
    - In the end, the root of T will be x.
  - R = subtree containing nodes larger than x, encountered on the access path towards x.
- The tree is descended two levels at a time.

Splay trees: top-down splaying

- In the figures that follow, the access path is indicated by a thicker line.
- The tree is descended two levels at a time. We are interested in the current node, its child and its grandchild along the access path.
- There are three cases:
  - zig
    - The target node is the child of the current node
  - zig-zig
    - The three nodes of interest form a straight line
  - zig-zag
    - The three nodes of interest form a zig-zag line

Splay trees: top-down splaying



We are splaying node 'a'.

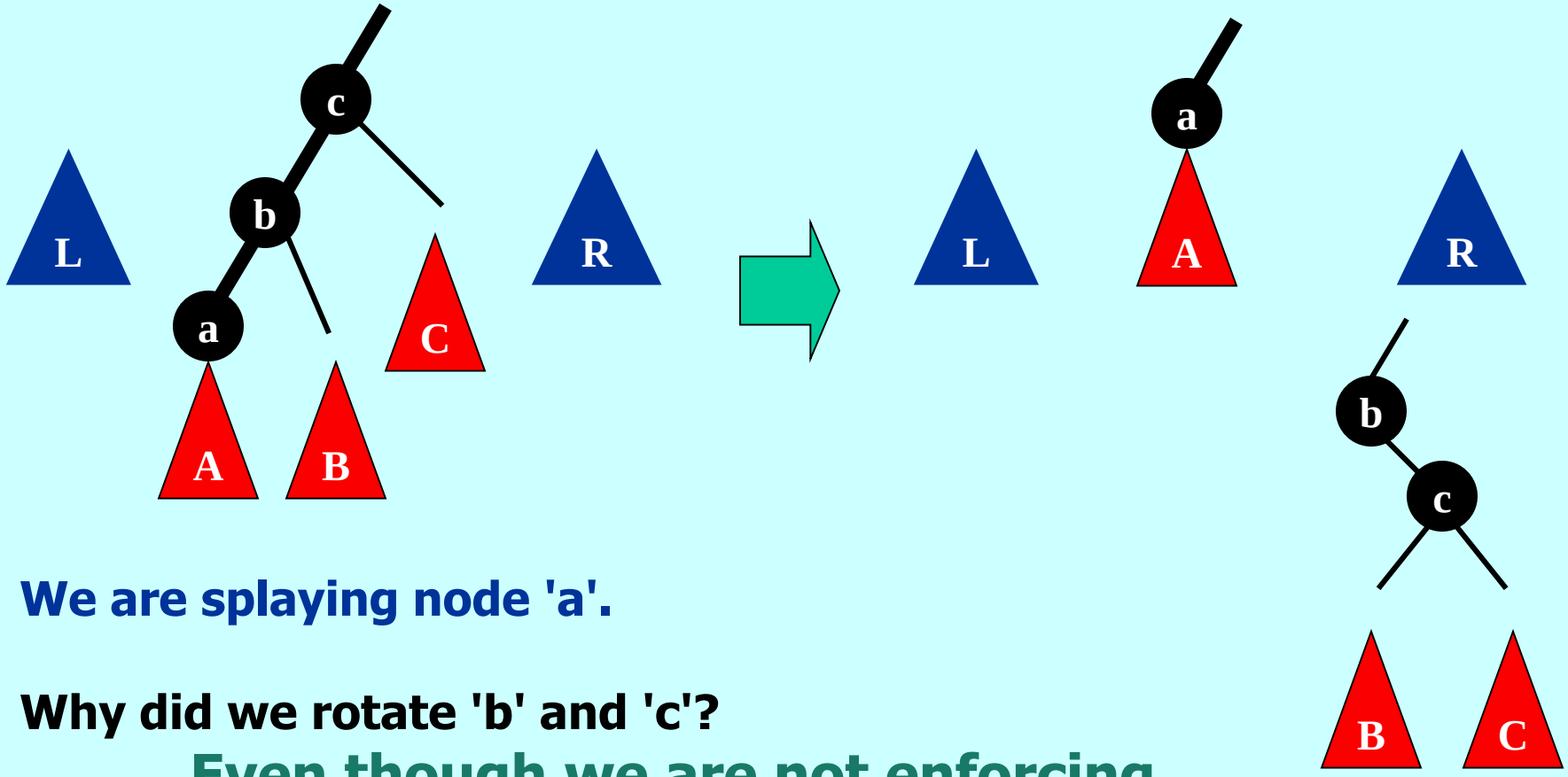
Where should 'b' go?

'b' is greater than 'a', so it is attached to R

Where in R should 'b' be attached?

'b' is smaller than every other node in R, so it should be placed in the lower left.

Splay trees: top-down splaying

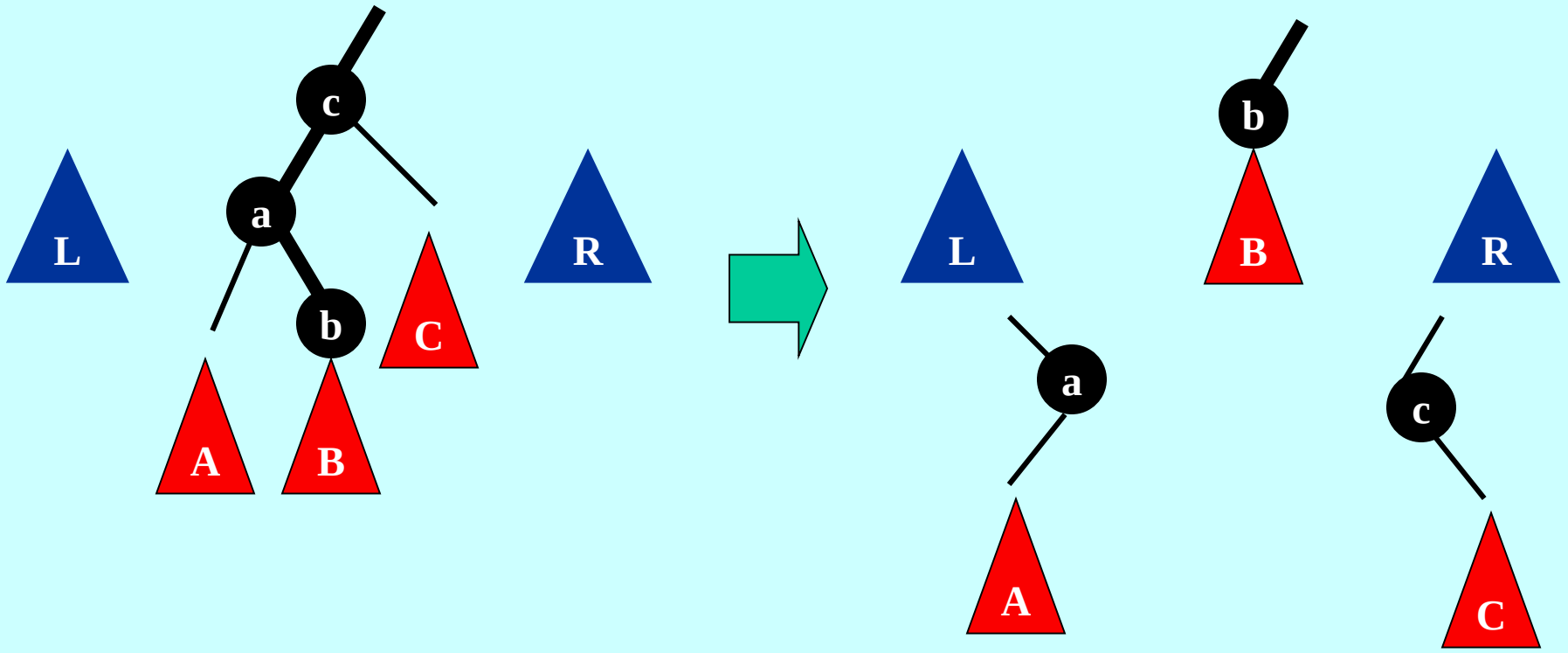


We are splaying node 'a'.

Why did we rotate 'b' and 'c'?

Even though we are not enforcing any balance restrictions, we still want to avoid a very imbalanced tree. The rotation helps in that (think of how R would look if we never rotated.)

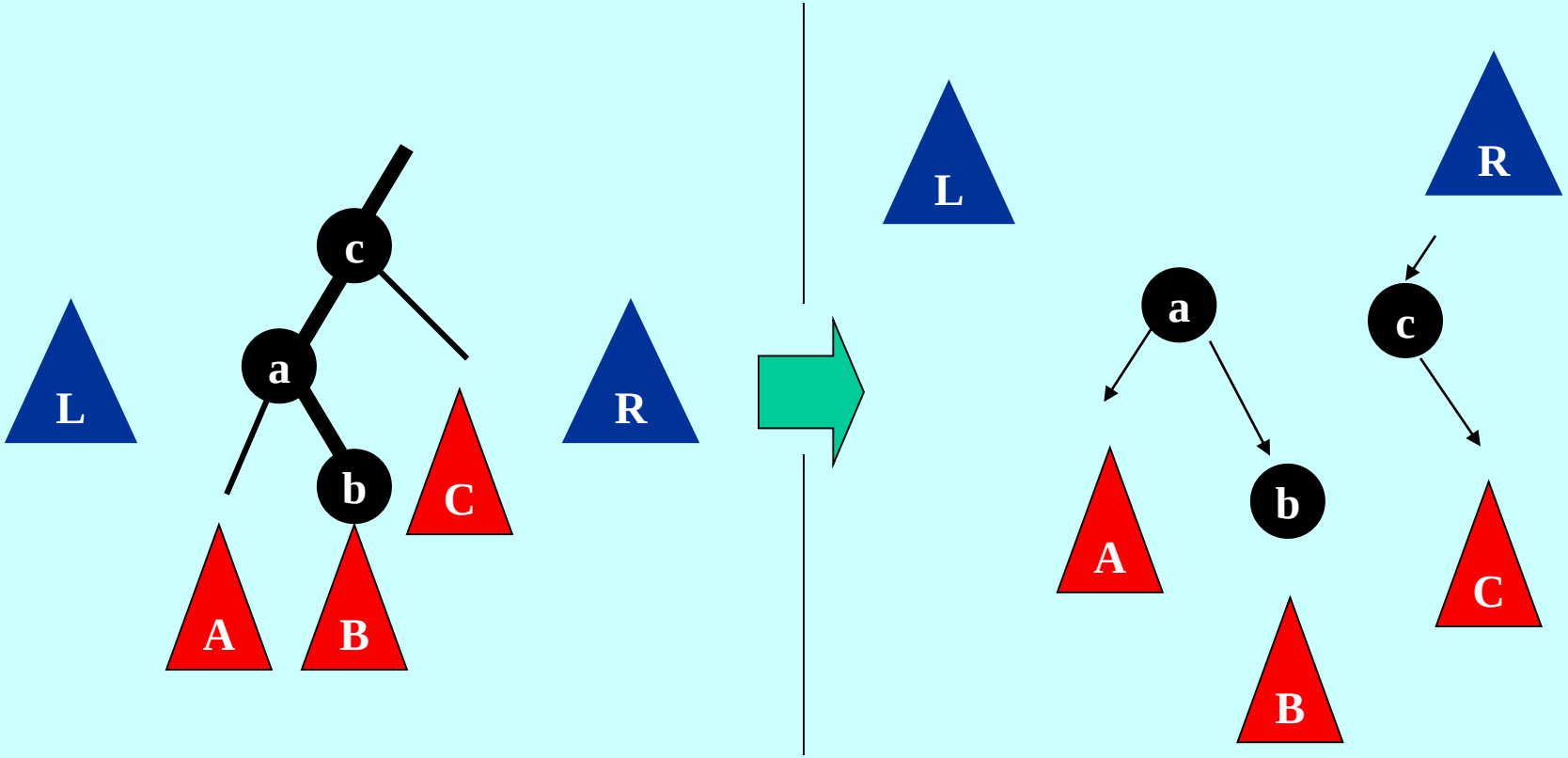
Splay trees: top-down splaying



We are splaying node 'b'.

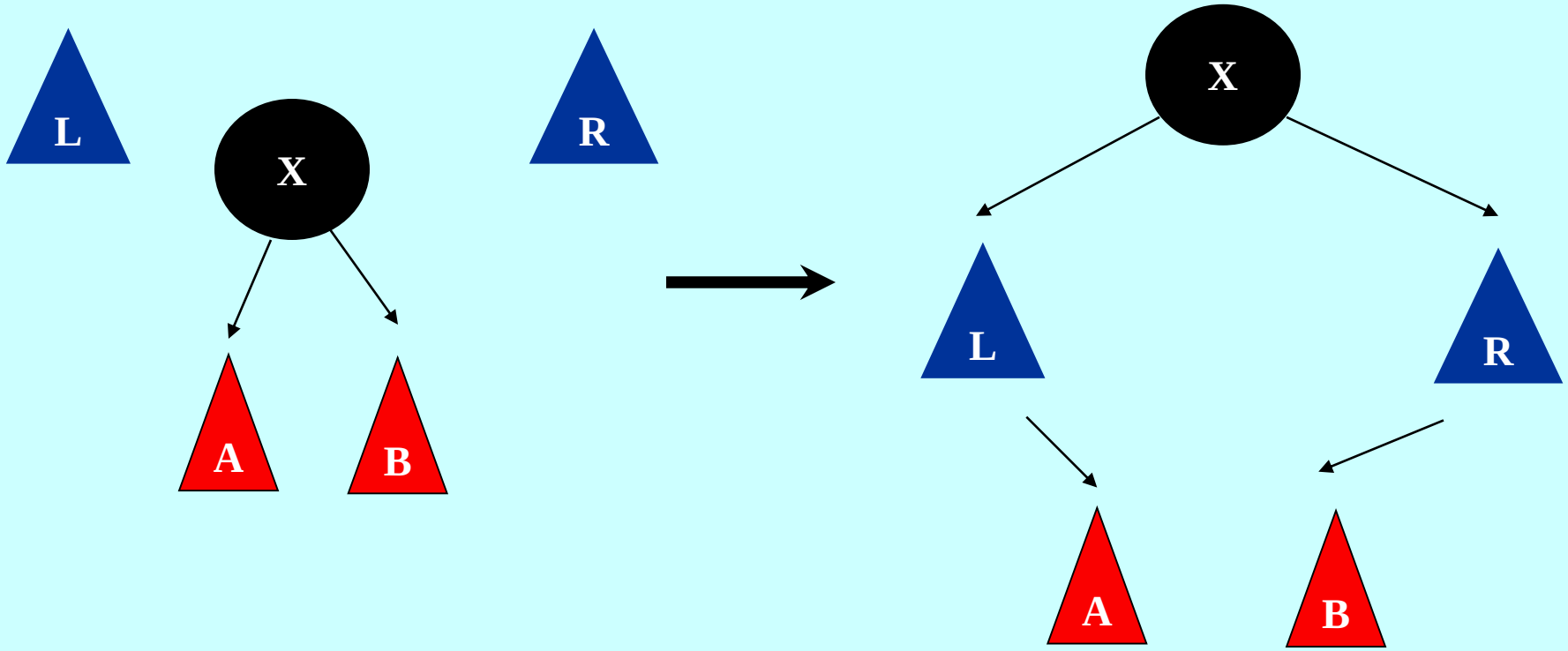
'a' is smaller than 'b', and is placed in L
'c' is greater than 'b', and is placed in R

Zig-Zag (Simplified)



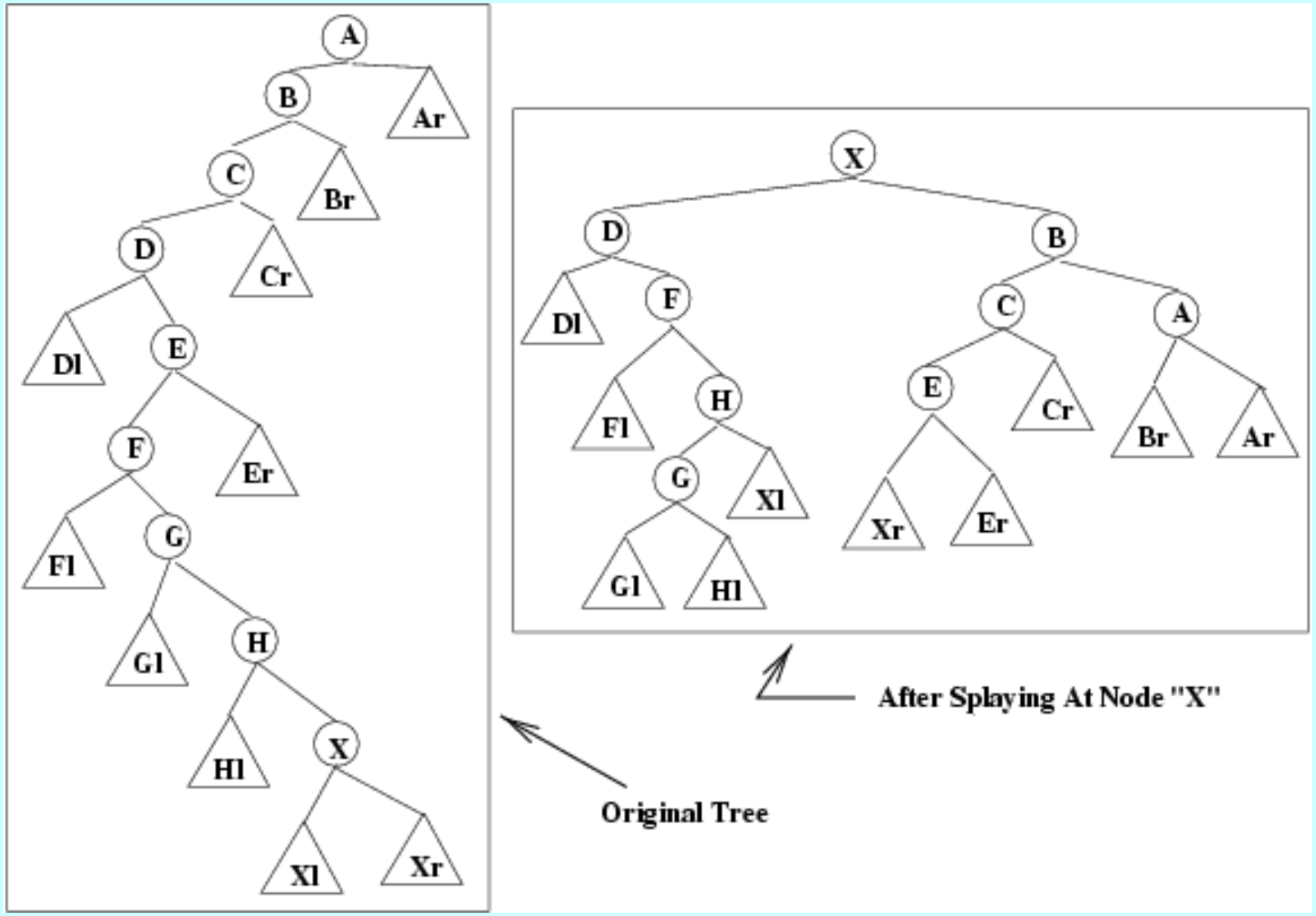
The value to be splayed is in the tree rooted at b. To make code simpler, the Zig-Zag rotation is reduced to a single Zig. This results in more iterations in the splay process.

Reassembling the Splay Tree

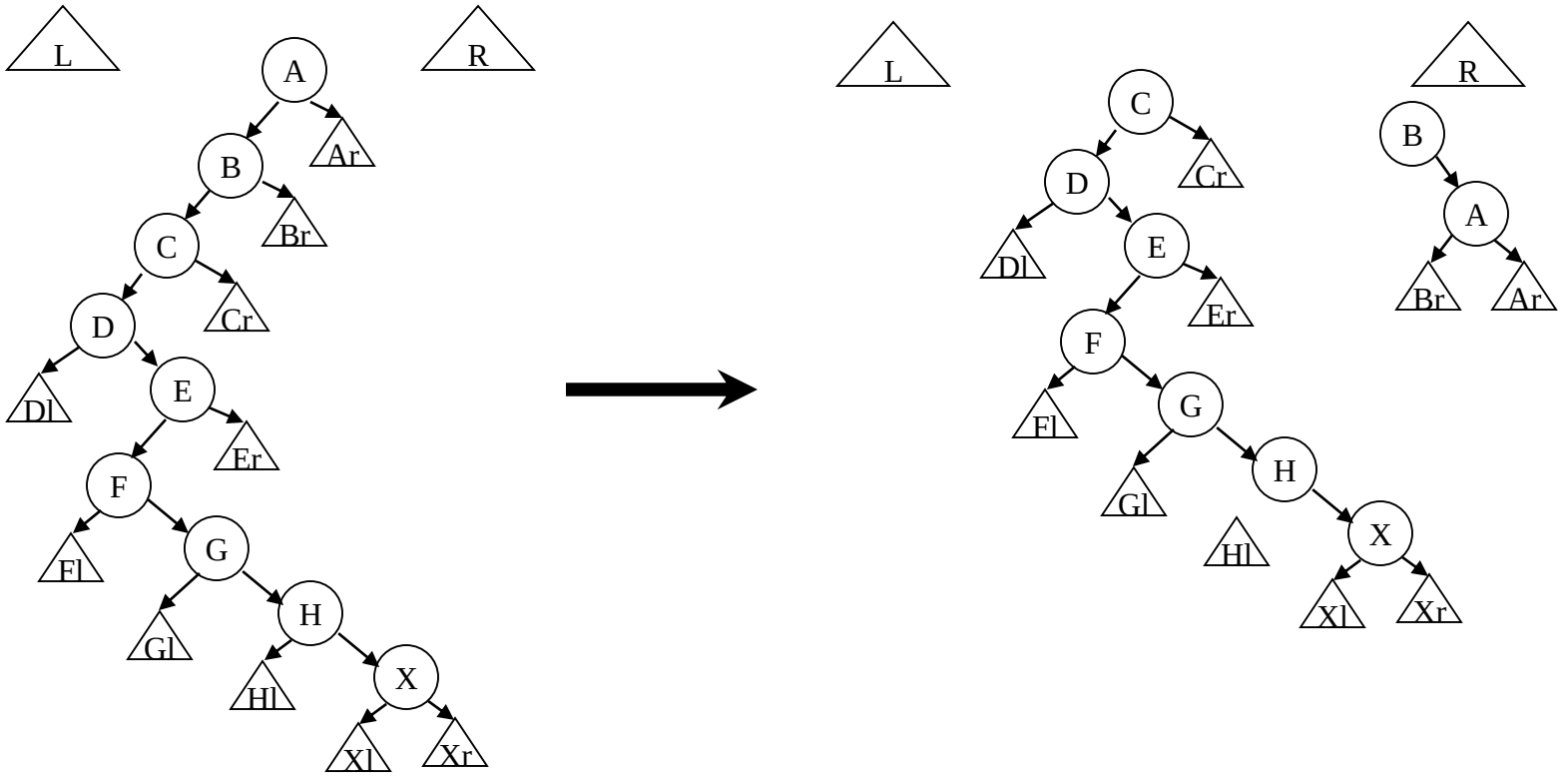


When the value to be splayed to the root is at the root of the "center" tree, we have reached the point where we are ready to reassemble the tree. This is accomplished by a) making XL the right child of the maximum element in L, b) making XR the left child of the minimum element in R, and then making L and R the left and right children of X

Example: (from bottom-up)



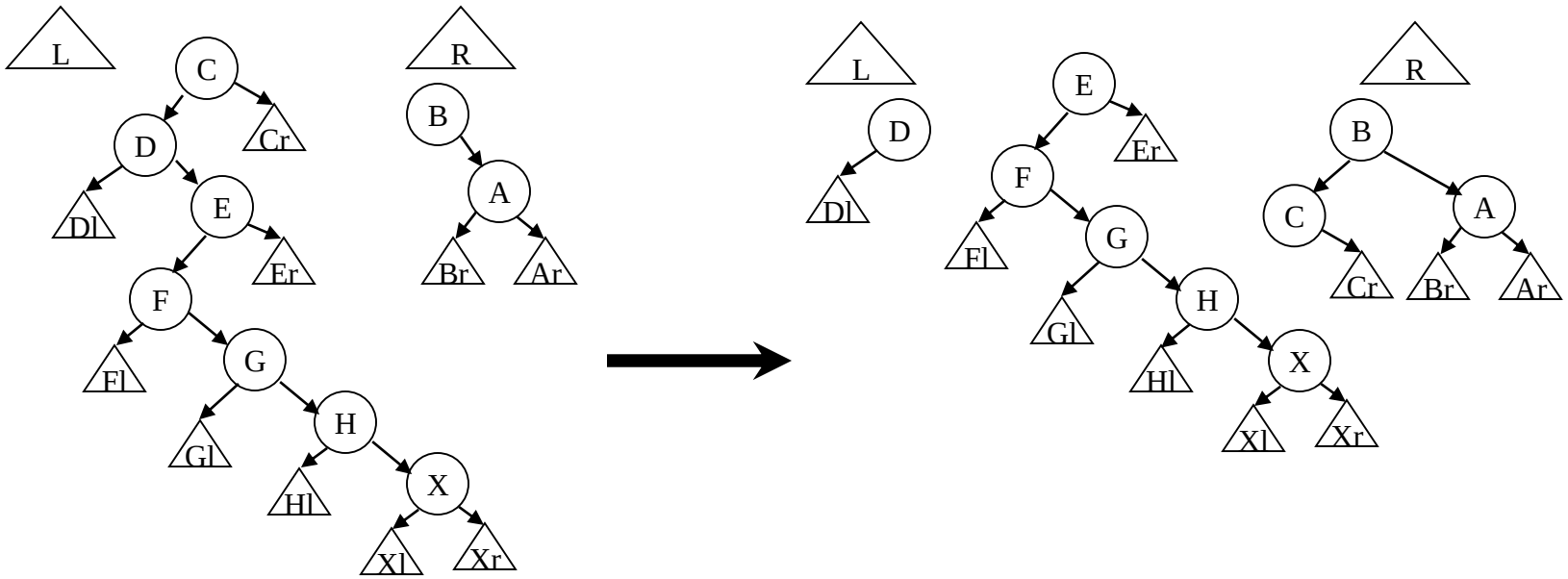
Operation 1: Zig-Zig



Rotate B around A and make L child of minimum element in R (which is now empty)

L is still empty, and R is now the tree rooted at B. Note that R contains nodes > X but not in the right subtree of X.

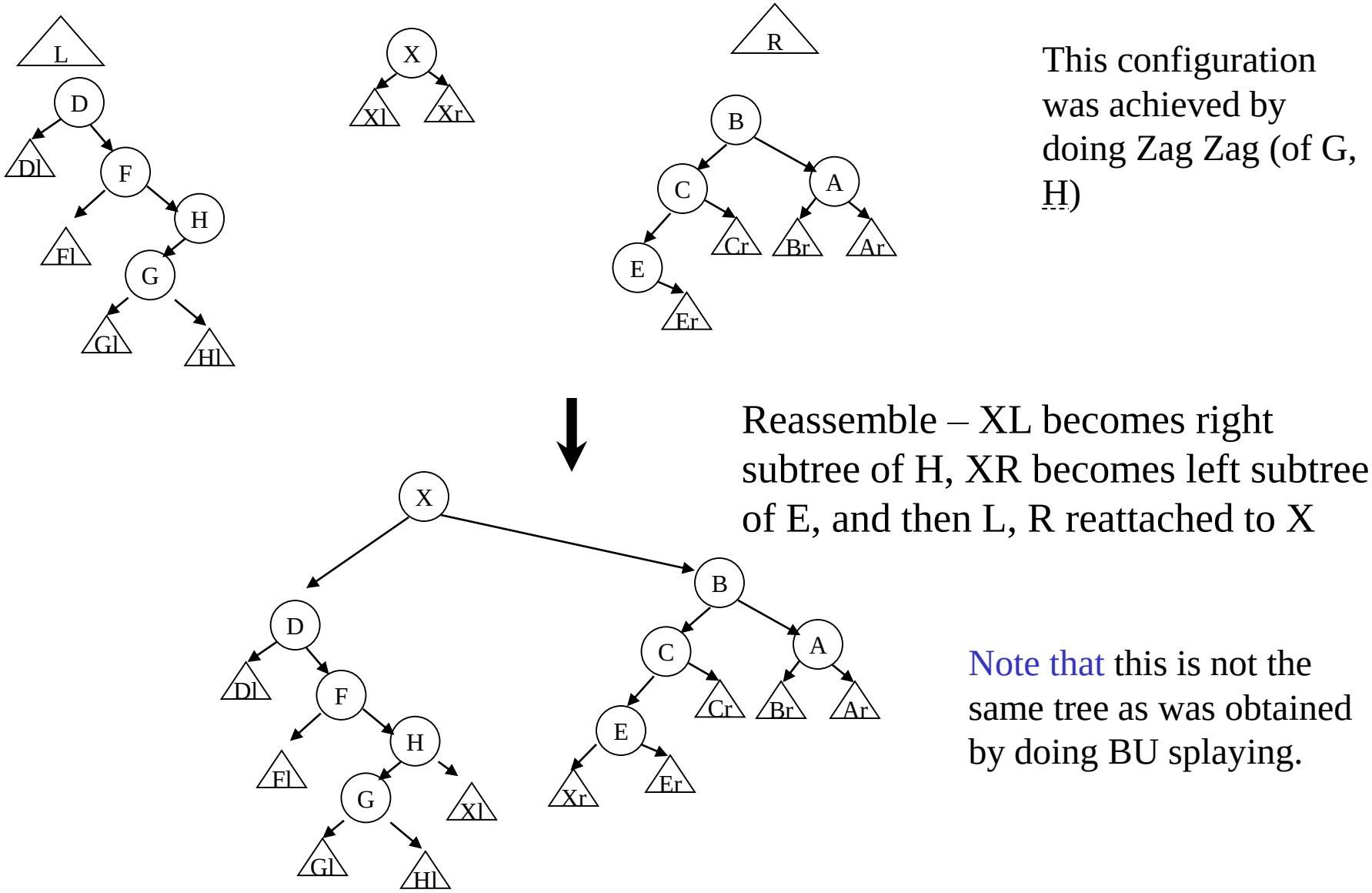
Operation 2: Zig-Zag



Just perform Zig
(simplified Zig-Zag)

L was previously empty and it now consists of node D and D's left subtree

After X reaches root:

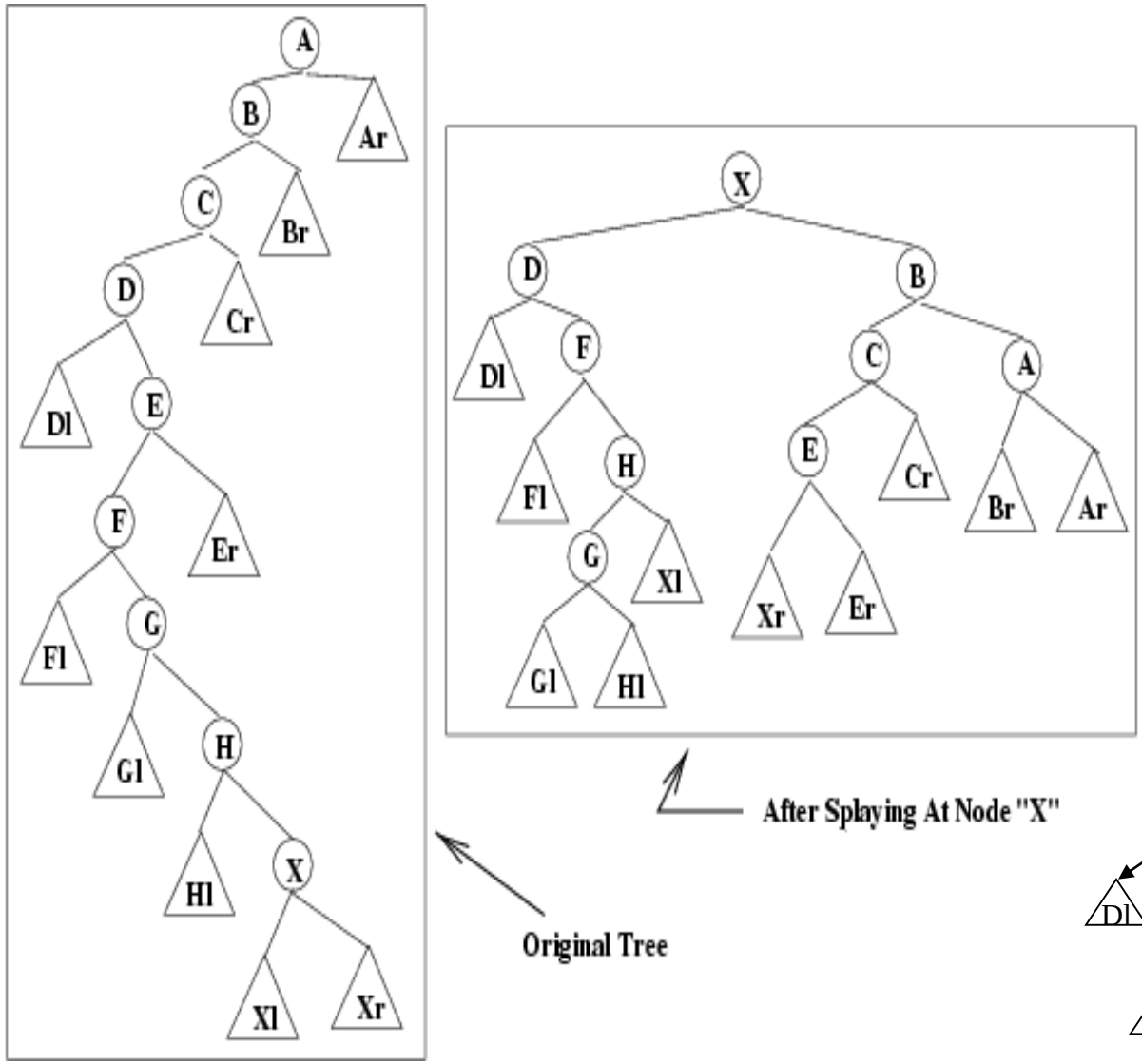


This configuration was achieved by doing Zag Zag (of G, H)

Reassemble – XL becomes right subtree of H, XR becomes left subtree of E, and then L, R reattached to X

Note that this is not the same tree as was obtained by doing BU splaying.

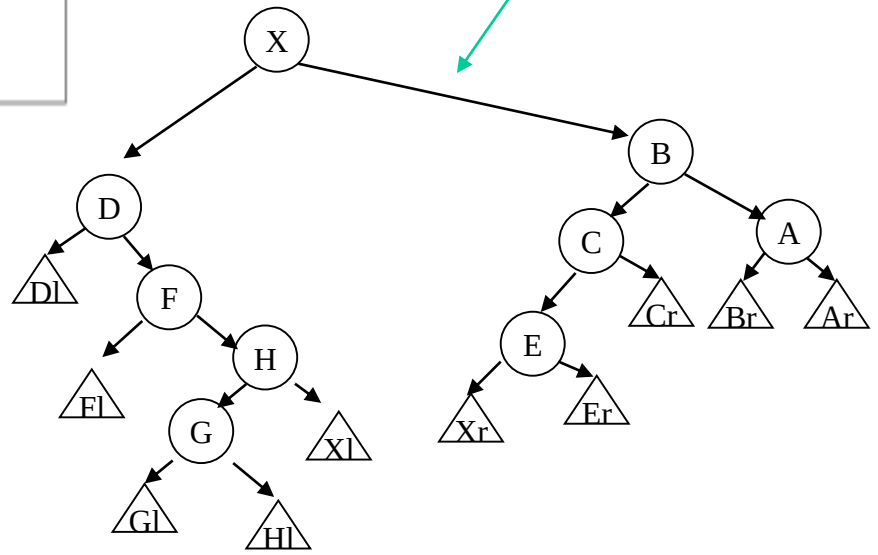
Example: bottom-up vs top-down



Original Tree

After Splaying At Node "X"

top-down



Reference

- *Daniel Dominic Sleator and Robert Endre Tarjan. 1985. Self-adjusting binary search trees. J. ACM 32, 3 (July 1985), 652–686. DOI:https://doi.org/10.1145/3828.3835*
- *Data Structures and Algorithm Analysis in C++* Section 4.5 and 12.1
- *http://users.cis.fiu.edu/~weiss/dsaa_c++3/code/SplayTree.h*

Next

- B-树
- 数据结构（C++语言版）第三版 Chapter 8.2