局部性

❖ Locality：　刚被访问过的数据，极有可能很快地再次被访问 时间局部性.

这一现象在信息处理过程中屡见不鲜，比如...

❖ BST：　刚被访问过的节点，极有可能很快地再次被访问 空间局部性

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$\mathcal{O}(m \cdot \log n)$时间

❖ 利用局部性，能否更快？

❖ 自适应链表：　节点一旦被访问，随即移动到最前端

❖ BST策略：　节点一旦被访问，随即调整到树根 SPLAY伸展.

❖ 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

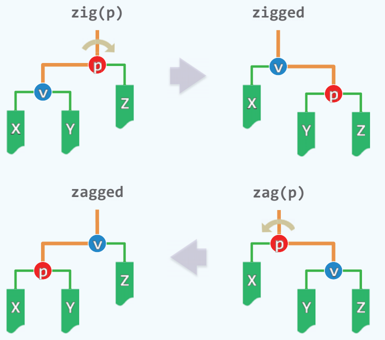
随即推送至根

❖ 与其说"推"，不如说"爬"

一步一步往上爬

❖ 自下而上，逐层旋转

- zig(v->parent)
- zag(v->parent)



实例

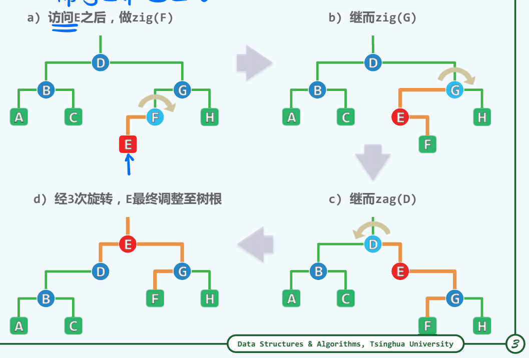
❖ 伸展过程的效率

是否足够地高？

❖ 这取决于

- 树的初始形态和
- 节点的访问次序

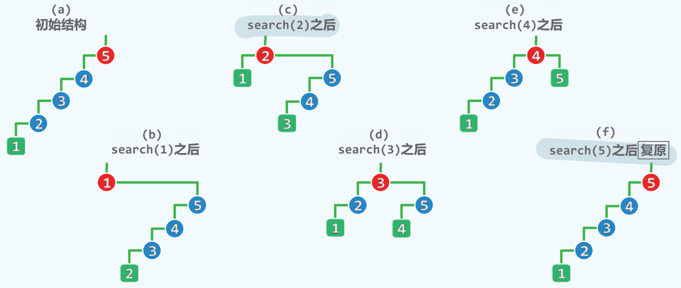
需将E推送至根



最坏情况

：树退化为链表，且每次访问最深位置.

❖ 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$\Omega(n)$！

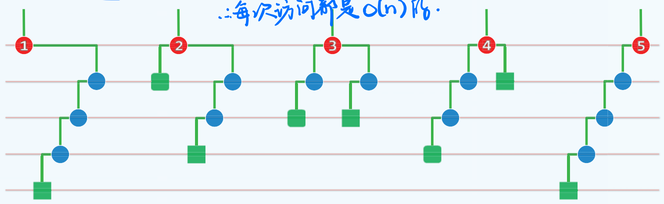


问题出在...

❖ 全树拓扑始终呈单链条结构，等价于一维列表

被访问节点的深度，呈周期性的算术级数演变：{ n-1, n-2, n-3, ..., 3, 2, 1 }

∴每次访问都是O(n)阶.



❖ 问题的症结既已确定，便可针对性地改进...

08-A2

8.高级搜索树

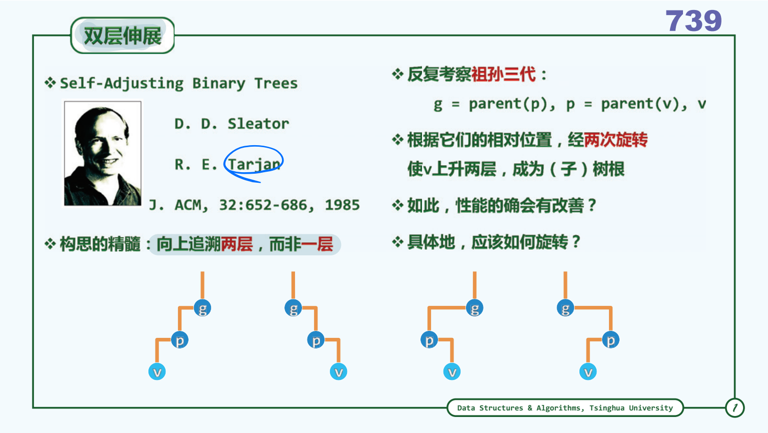
伸展树

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



739

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：

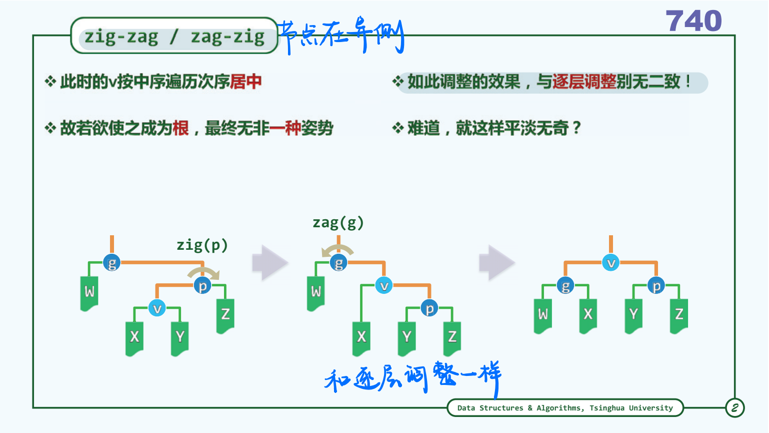
g = parent(p), p = parent(v), v

❖ 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？



740

zig-zag / zag-zig

节点在异侧

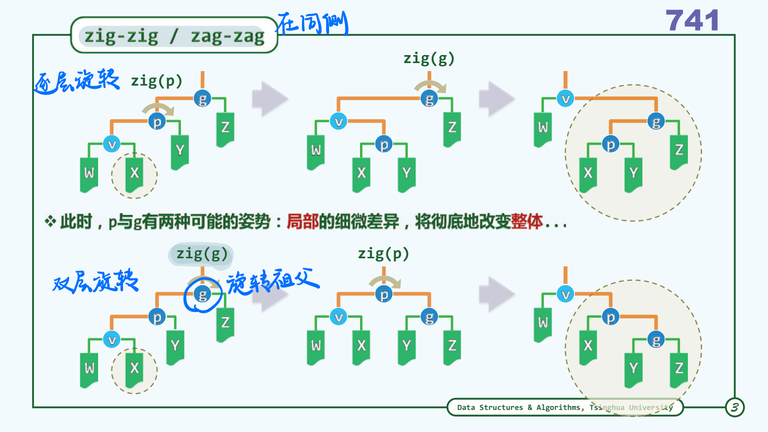
❖ 此时的v按中序遍历次序居中

❖ 故若欲使之成为根，最终无非一种姿势

❖ 如此调整的效果，与逐层调整别无二致！

❖ 难道，就这样平淡无奇？

和逐层调整一样



741

zig-zig / zag-zag

在同侧

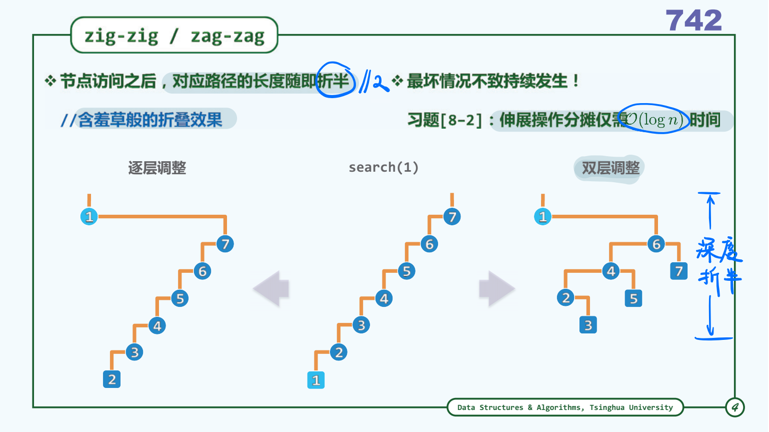
逐层旋转 zig(p)

zig(g)

❖ 此时，p与g有两种可能的姿势：局部的细微差异，将彻底地改变整体...

双层旋转 zig(g) 旋转祖父

zig(p)



742

zig-zig / zag-zag

❖ 节点访问之后，对应路径的长度随即折半 /2

//含羞草般的折叠效果

❖ 最坏情况不致持续发生！

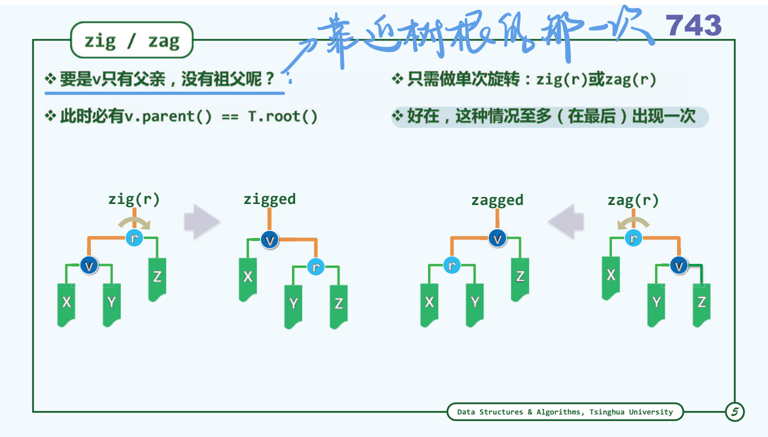
习题[8-2]：伸展操作分摊仅需$\mathcal{O}(\log n)$时间

逐层调整

search(1)

双层调整

深度折半



743

zig / zag

靠近树根的那一次

❖ 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

744

08-A3

8.高级搜索树

伸展树

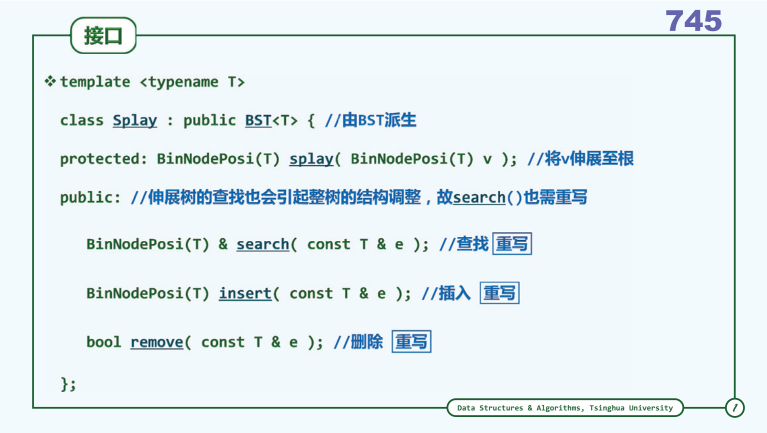
算法实现

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。

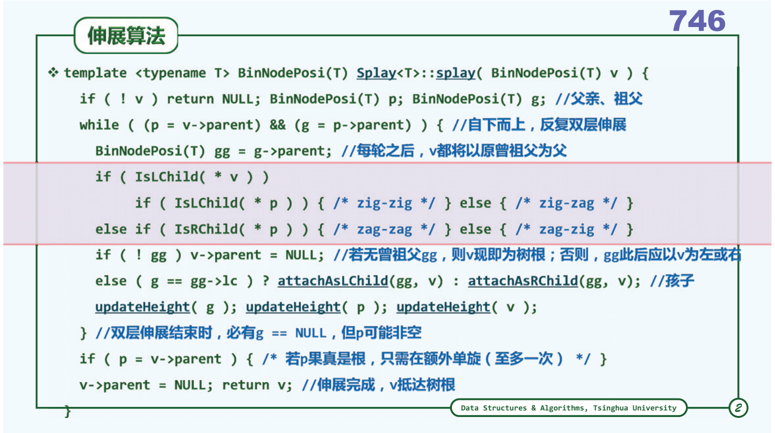
邓俊辉

deng@tsinghua.edu.cn



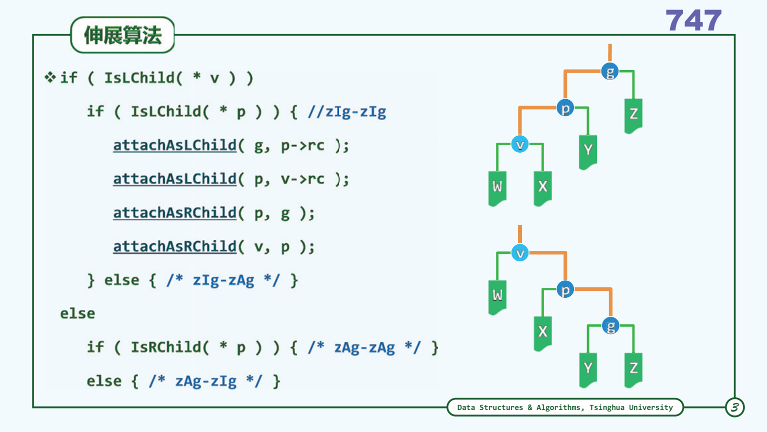
接口

```
❖ 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
  };
```



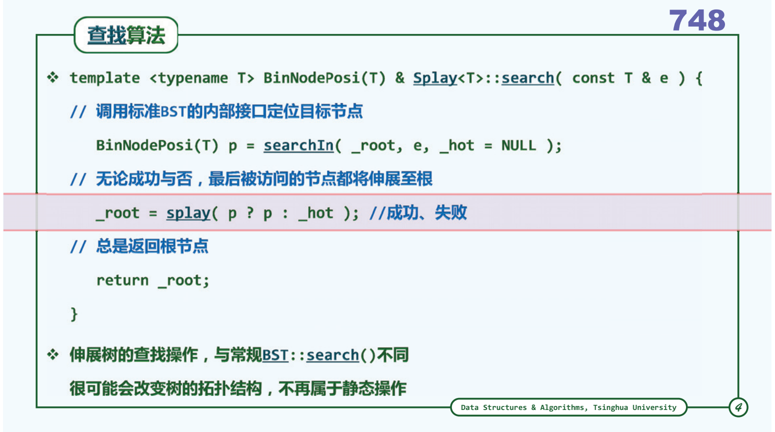
伸展算法

```
❖ 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 */ }
```

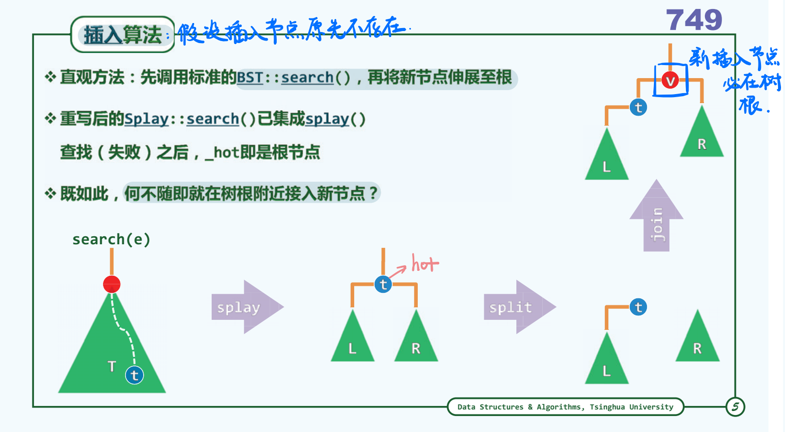


查找算法

```
❖ 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

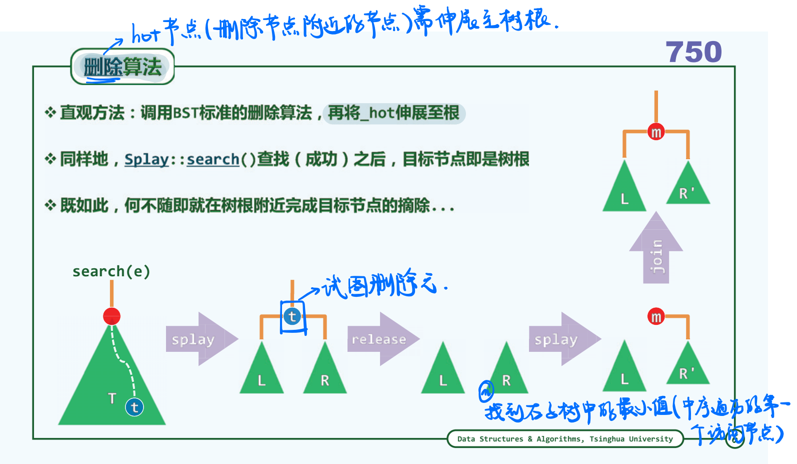
：假设插入节点原先不存在.

❖ 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？



hot节点(删除节点附近的节点)需伸展至树根.

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...