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

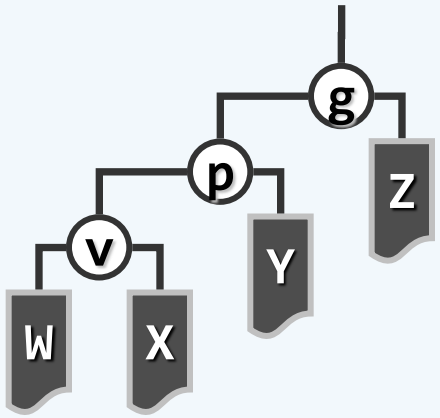
```
template <typename T>
class Splay : public BST<T> { //由BST派生
protected: BinNodePosi(T) splay( BinNodePosi(T) v ); //将v伸展至根
public: //伸展树的查找也会引起整树的结构调整，故search()也需重写
   BinNodePosi(T) & search( const T & e ); //查找 重写
   BinNodePosi(T) insert( const T & e ); //插入 重写
   bool remove( const T & e ); //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

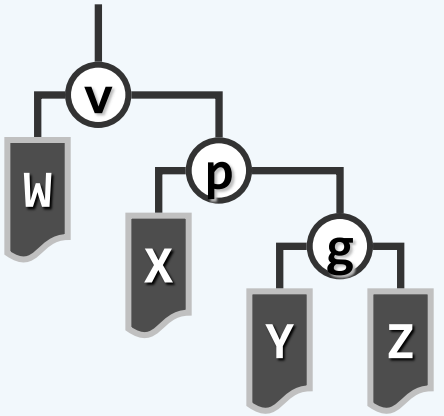
伸展算法

```
❖ 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

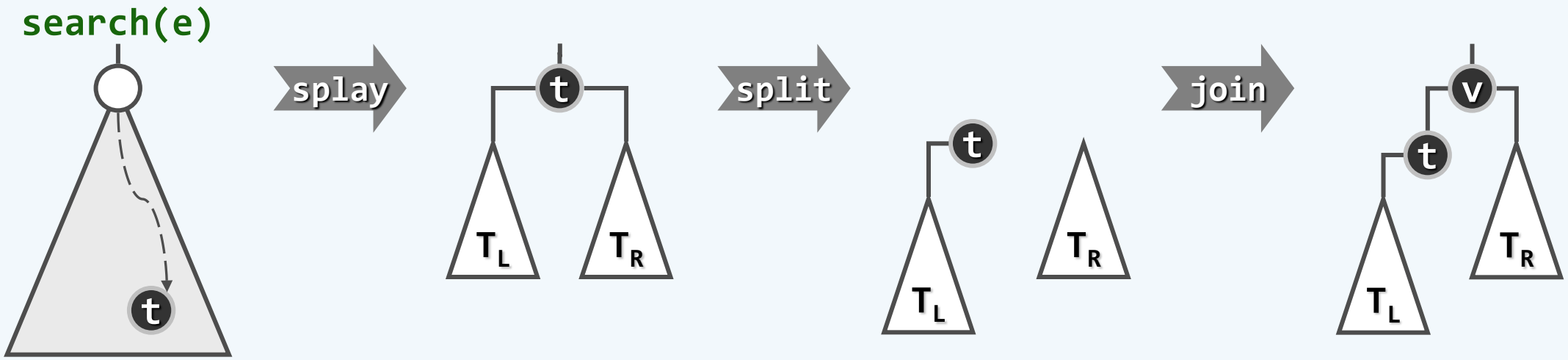
- 直观方法：调用BST标准的插入算法，再将新节点伸展至根

  其中，首先需调用BST::search()

- 重写后的Splay::search()已集成了splay()操作

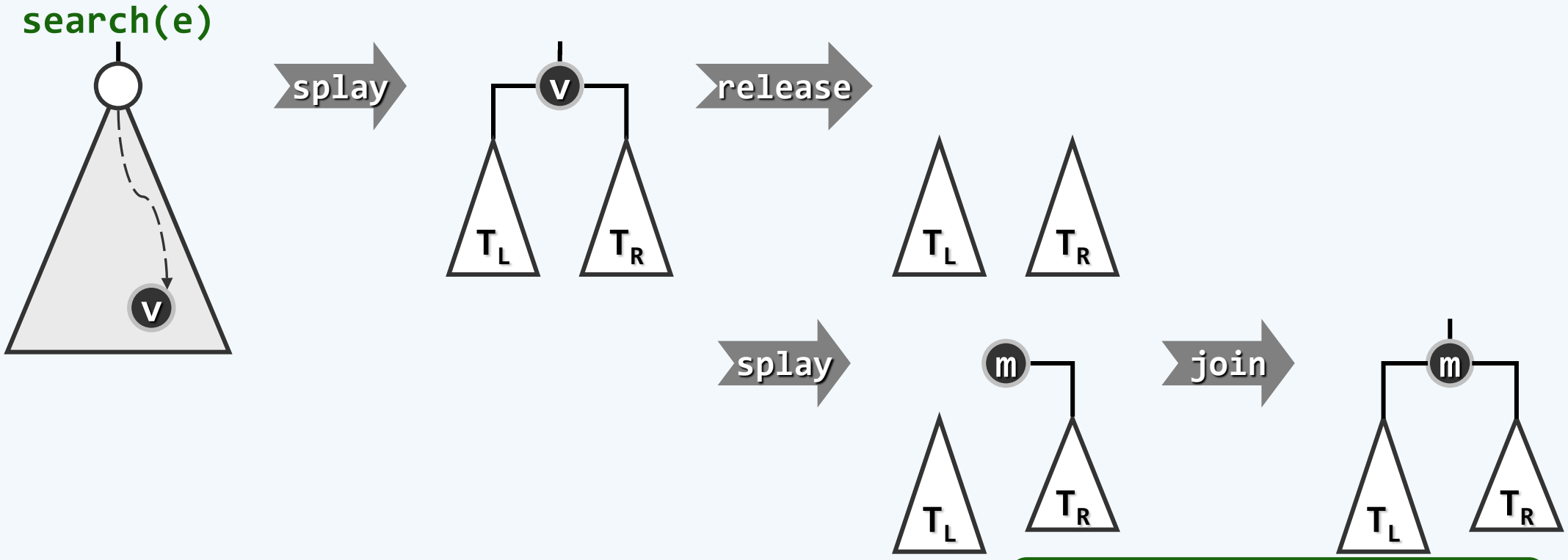
  查找（失败）之后，_hot即是根节点

- 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

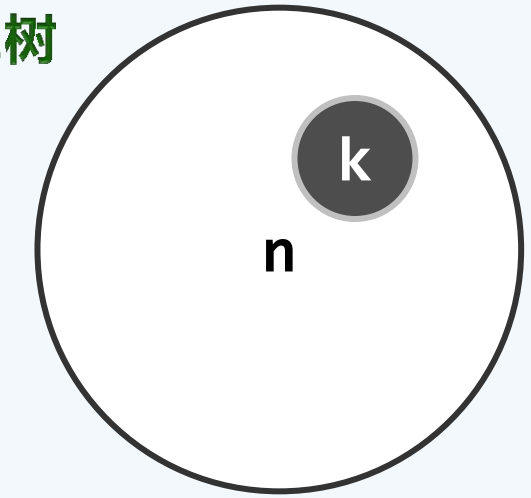
- 无需记录节点高度或平衡因子；编程实现简单易行——优于AVL树

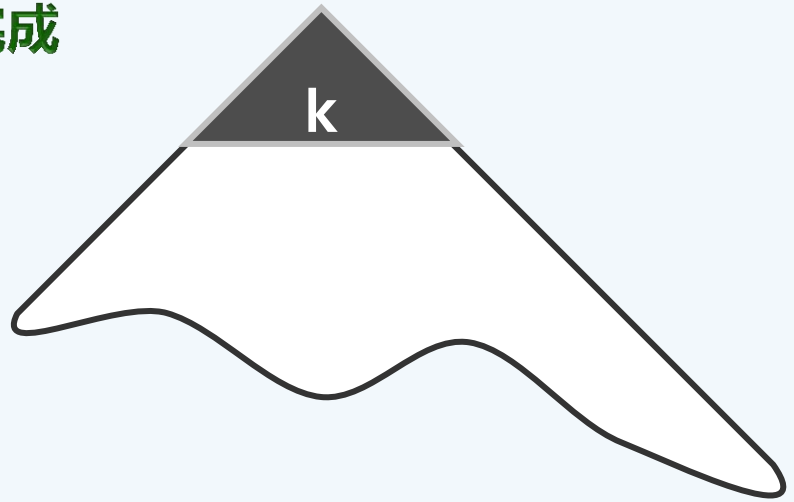
  分摊复杂度$O(\log n)$——与AVL树相当

- 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