08-A3

高级搜索树

伸展树：算法实现

邓俊辉

deng@tsinghua.edu.cn

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

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：总体思路

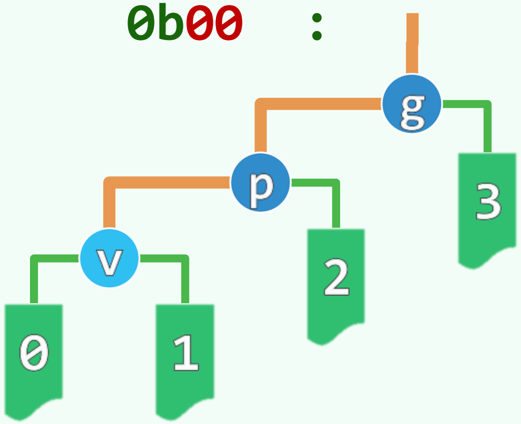
```
template <typename T> BinNodePosi<T> Splay<T>::splay( BinNodePosi<T> v ) {
   BinNodePosi<T> p, g; //父亲、祖父
   while ( (p = v->parent) && (g = p->parent) ) {
      BinNodePosi<T> gg = g->parent; //great-grand parent
      switch ( ( IsLChild( p ) << 1 ) | IsLChild( v ) ) {
         /* 视p、v的拐向分四种情况，相应地双层伸展 */
      }
      /* 向上联接，更新高度 */
   }
   if ( p = v->parent ) { /* 若p果真是根，只需再额外单层伸展一次 */ }
   v->parent = NULL; return v; //伸展完成，v抵达树根
}
```

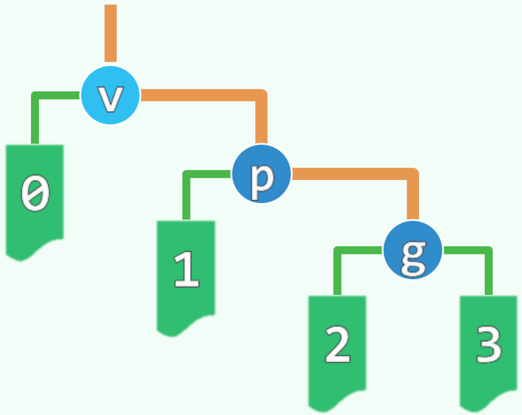
伸展算法：双层伸展（1/2）

```
while ( (p = v->parent) && (g = p->parent) ) { //自下而上，反复双层伸展
   BinNodePosi<T> gg = g->parent; //great-grand parent
   switch ( ( IsRChild( p ) << 1 ) | IsRChild( v ) ) { //视p、v拐向，分四种情况
      case 0b00   : /* ... zig-zig ... */
      case 0b01   : /* ... zig-zag ... */
      case 0b10   : /* ... zag-zig ... */
      default     : /* ... zag-zag ... */ /*0b11*/
   }
   if ( !gg ) v->parent = NULL; //若原曾祖父*gg不存在，则*v现应为树根；否则*gg应以
   else ( g == gg->lc ) ? gg->attachLc(v) : gg->attachRc(v); //v作为左或右孩子
   g->updateHeight(); p->updateHeight(); v->updateHeight();
}
```

伸展算法：双层伸展（2/2）

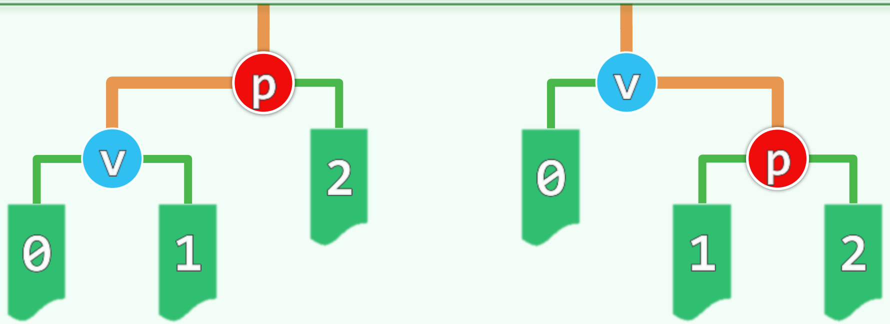
```
switch ( ( IsRChild( p ) << 1 ) | IsRChild( v ) ) { //视p、v拐向，分四种情况
   case    0b00  :            /*2*/ g->attachLc( p->rc );
                              /*1*/ p->attachLc( v->rc );
                                    p->attachRc( g );
                                    v->attachRc( p );
                                    break; //zig-zig
   case    0b01  : p->attachRc( v->lc ); g->attachLc( v->rc ); //zig-zag
                   v->attachRc( g     ); v->attachLc( p     );   break;
   case    0b10  : p->attachLc( v->rc ); g->attachRc( v->lc ); //zag-zig
                   v->attachLc( g     ); v->attachRc( p     );   break;
   default/*0b11*/: g->attachRc( p->lc ); p->attachRc( v->lc ); //zag-zag
                   p->attachLc( g     ); v->attachLc( p     );   break;
}
```





伸展算法：单层伸展（至多一次）

```
template <typename T> BinNodePosi<T> Splay<T>::splay( BinNodePosi<T> v ) {
   BinNodePosi<T> p, g; //父亲、祖父
   while ( (p = v->parent) && (g = p->parent) ) { /* 双层伸展 */ }
   if ( p = v->parent ) { //若p果真是根，只需再额外单层伸展一次
      if ( IsLChild( v ) ) {
         p->attachLc( v->rc );
         v->attachRc( p );
      } else { p->attachRc( v->lc ); v->attachLc( p ); }
      p->updateHeight(); v->updateHeight();
   }
   v->parent = NULL; return v; //伸展完成，v抵达树根
}
```



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 );
    //无论如何，最后被访问的节点都将伸展至根
    _root = p ? splay(p) : _hot ? splay(_hot) : NULL; //成功、失败、空树
    //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

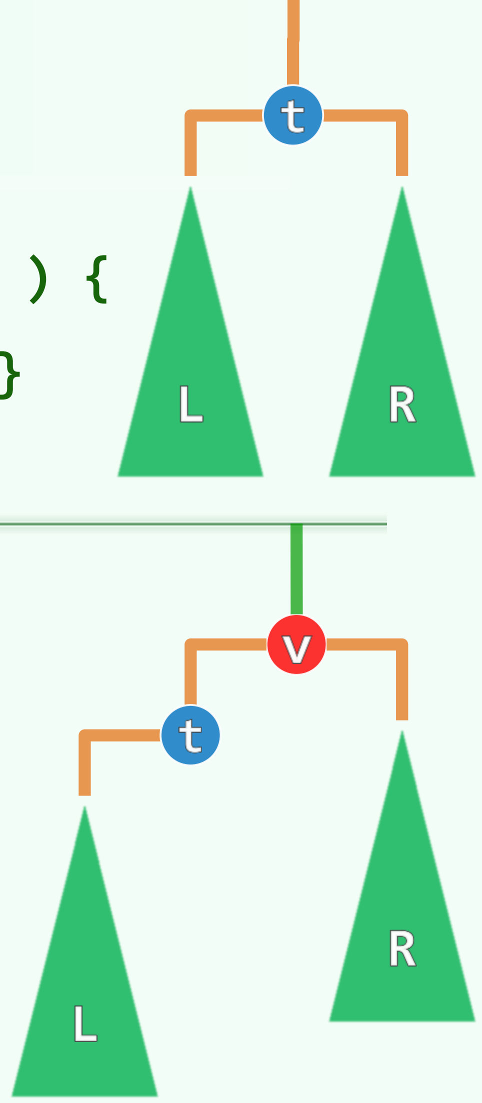
❖ 直观方法：先调用标准的BST::search()，再将新节点伸展至根

❖ Splay::search()已集成splay()，查找失败之后，_hot即是根

❖ 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 ); }
   BinNodePosi<T> t = search( e ); if ( e == t->data ) return t;
   if ( t->data < e ) {  //在右侧嫁接（rc或为空，lc == t必非空）
      _root = t->parent = new BinNode<T>( e, NULL, t, t->rc );
      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 );
      t->lc = NULL;
   }
   _size++; t->updateHeightAbove(); return _root; //更新记录，插入成功
} //无论如何，返回时总有_root->data == e
```



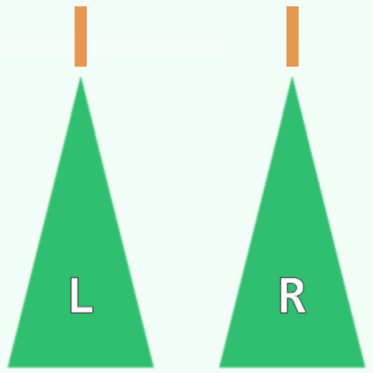
删除算法

❖ 直观方法：调用BST标准的删除算法，再将_hot伸展至根

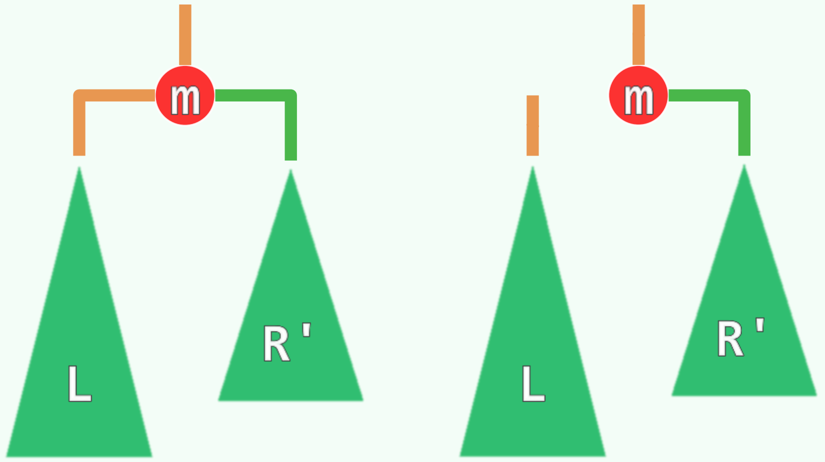
❖ 注意到，Splay::search()成功之后，目标节点即是树根

❖ 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;
   BinNodePosi<T> L = _root->lc, R = _root->rc; delete _root; //删
   if ( !R ) { if ( L ) L->parent = NULL; _root = L; //若R空，则L即是余树
   } else { //否则
      _root = R; R->parent = NULL;
      search( e ); //查找必败，但最小节点必伸展至根
      _root->attachLc(L); //可令其以L作为左子树
   }
    _size--; if ( _root ) _root->updateHeight(); return true; //更新记录，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...