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

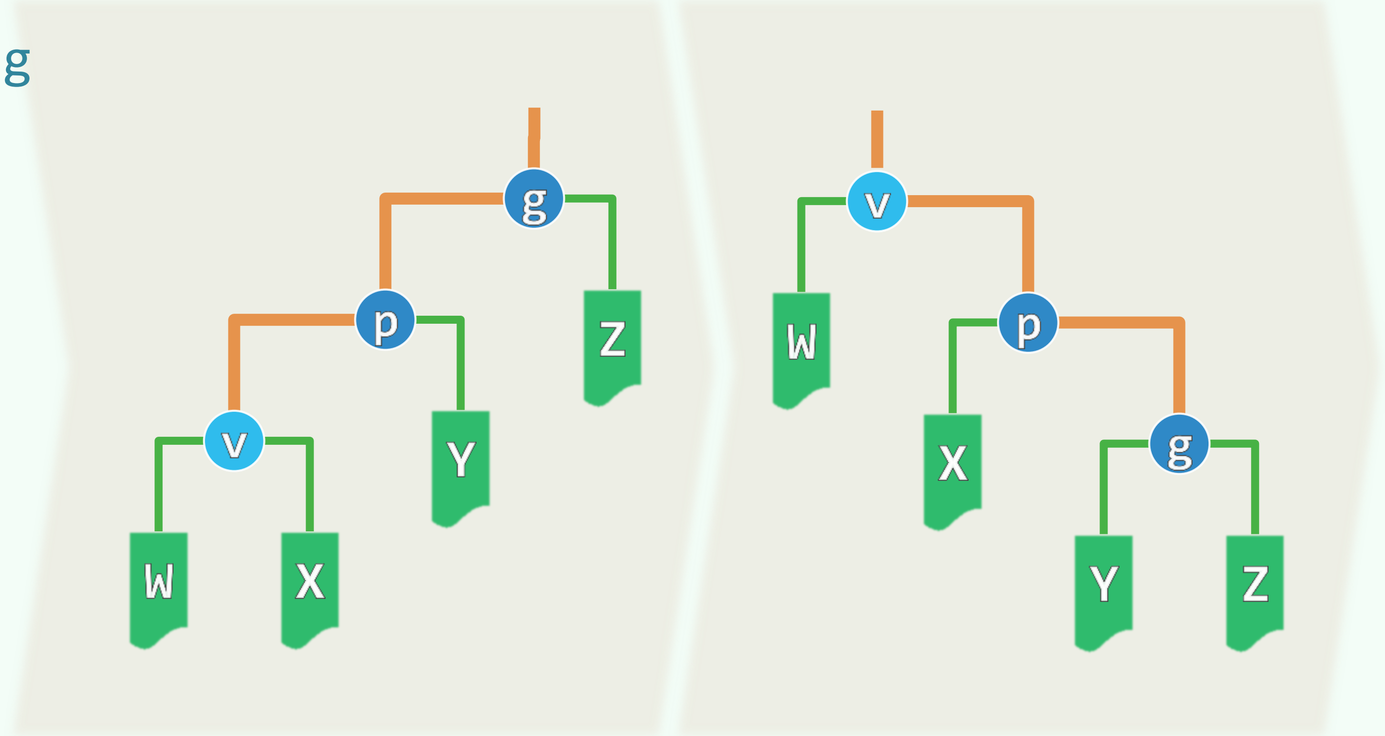
```
template <typename T> BinNodePosi<T> Splay<T>::splay( BinNodePosi<T> v ) {
   if ( ! v ) return NULL; BinNodePosi<T> p; BinNodePosi<T> g; //父亲、祖父
   while ( (p = v->parent) && (g = p->parent) ) {
      /* 自下而上，反复双层伸展 */
   }
   if ( p = v->parent ) { /* 若p果真是根，只需再额外单旋一次 */ }
   v->parent = NULL;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



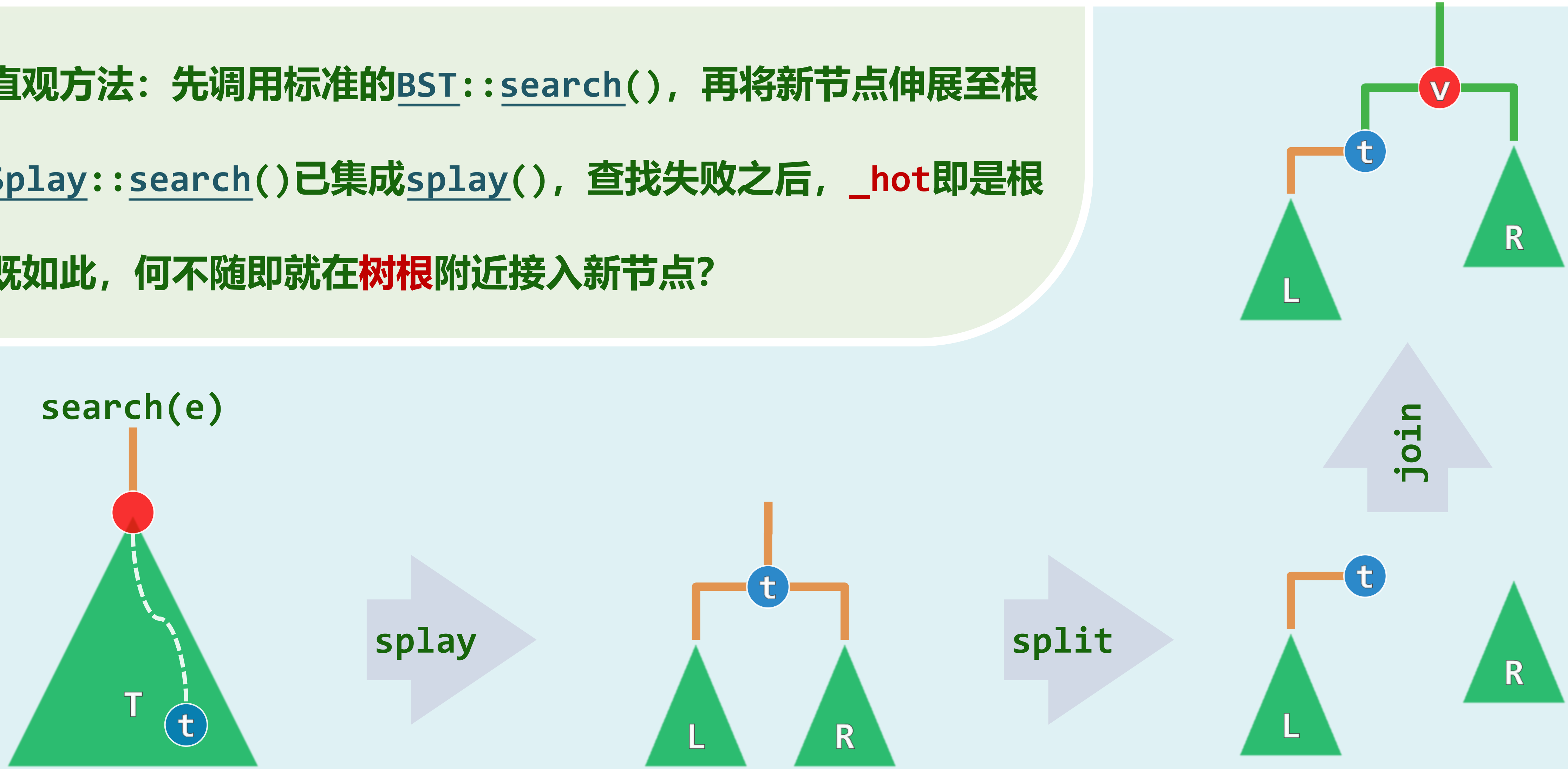
查找算法

❖
```
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
- ❖ 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

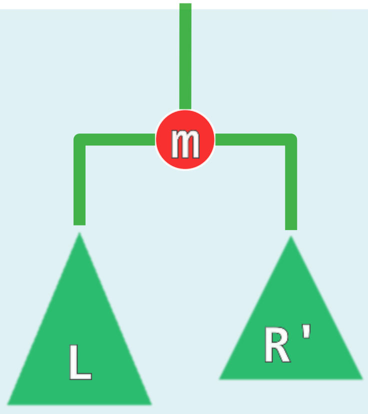
```
template <typename T> BinNodePosi<T> Splay<T>::insert( const T & e ) {
   if ( !_root ) { _size = 1; return _root = new BinNode<T>( e ); } //原树为空
   BinNodePosi<T> t = search( e ); if ( e == t->data ) return t; //t若存在，伸展至根
   if ( t->data < e ) {  //在右侧嫁接（rc或为空，lc == t必非空）
      t->parent = _root = 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
```

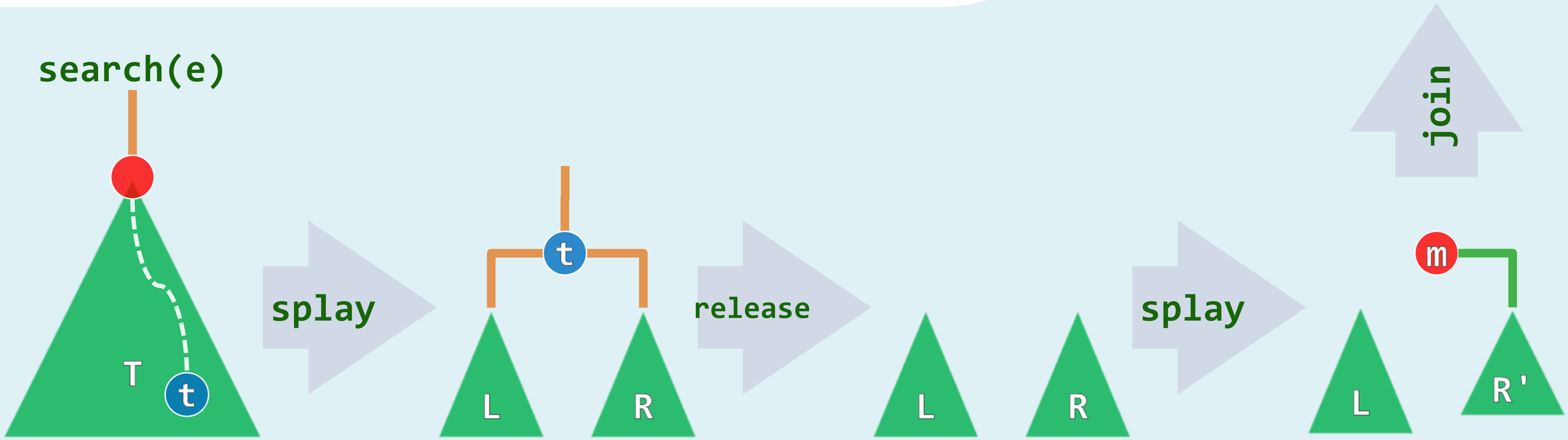
删除算法

❖ 直观方法：调用BST标准的删除算法，再将_hot伸展至根

❖ 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; //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...