

7. 二叉搜索树

(a) 概述

There's nothing in your head
the sorting hat can't see.
So try me on and I will tell
you where you ought to be.



- Harry Potter and the Sorcerer's Stone

邓俊辉

deng@tsinghua.edu.cn

查找

- ❖ 按照事先约定的规则，从数据集中找出符合**特定条件**的对象
- ❖ 对于算法的构建而言，属于最为基本而重要的静态操作
- ❖ 很遗憾，基本的数据结构并不能**高效**地**兼顾**静态查找与动态修改

基本结构	查找	插入/删除
无序向量	$O(n)$	$O(n)$
有序向量	$O(\log n)$	$O(n)$
无序列表	$O(n)$	$O(1)$
有序列表	$O(n)$	$O(n)$

- ❖ 那么，能否**综合**现有方法的优点？如何做到？

循关键码访问

❖ 数据项之间，依照各自的**关键码**彼此区分

call-by-key

❖ 条件：关键码之间支持

大小**比较**与

相等**比对**

❖ 数据集中的数据项

统一地表示和实现为词条**entry**形式



词条

❖ template <typename K, typename V> struct Entry { //词条模板类

K key; V value; //关键码、数值

Entry(K k = K(), V v = V()) : key(k), value(v) {}; //默认构造函数

Entry(Entry<K, V> const & e) : key(e.key), value(e.value) {}; //克隆

// 比较器、判等器 (从此, 不必严格区分词条及其对应的关键码)

bool operator< (Entry<K, V> const & e) { return key < e.key; } //小于

bool operator> (Entry<K, V> const & e) { return key > e.key; } //大于

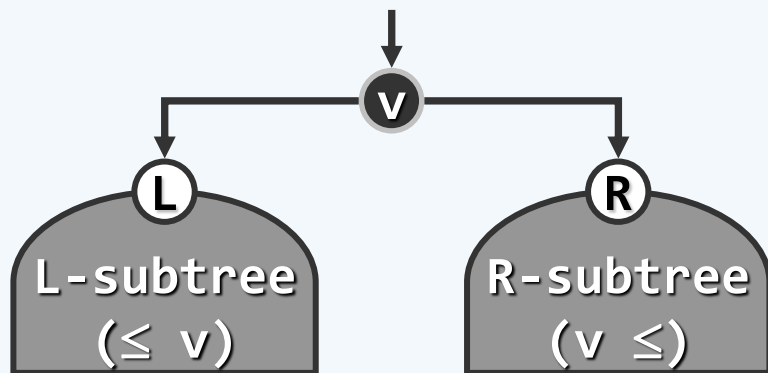
bool operator==(Entry<K, V> const & e) { return key == e.key; } //等于

bool operator!=(Entry<K, V> const & e) { return key != e.key; } //不等

};

顺序性

❖ **Binary Search Tree** : 节点 ~ 词条 ~ 关键码



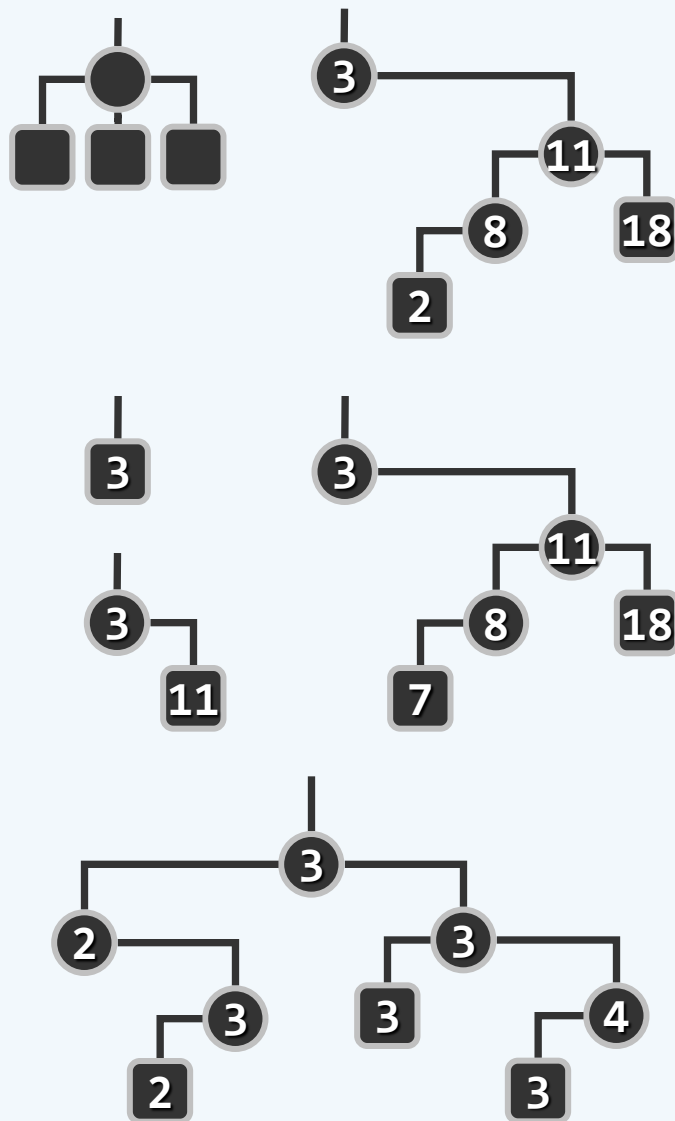
❖ **顺序性** : 任一节点均 **不小于** / **不大于** 其 **左** / **右** 后代

❖ **等效?** : 任一节点均不小于 / 不大于 其 **左** / **右** 孩子

❖ 为简化起见, 禁止重复词条

❖ 这种简化 : 应用中不自然, 算法上无必要

习题解析 : $[7-10] + [7-13] + [8-3]$



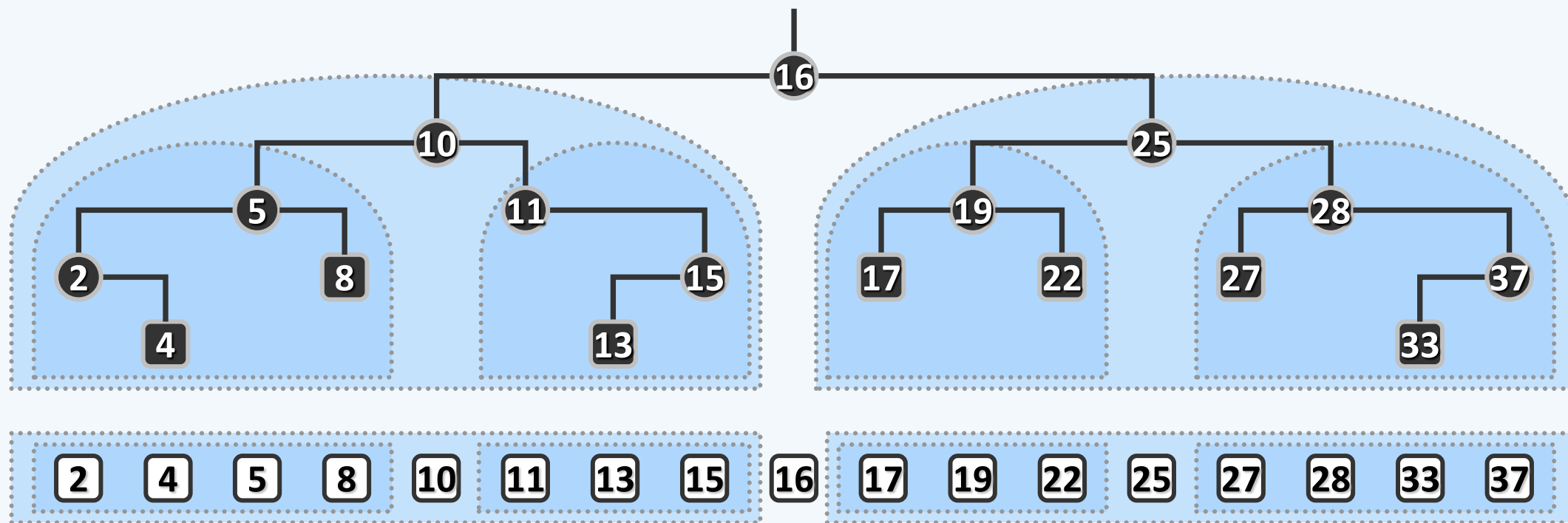
单调性

❖ 顺序性虽然只是对局部特征的刻画，但由此却可导出某种全局特征...

❖ 单调性：BST的中序遍历序列，必然单调非降

❖ 这一性质，也是BST的充要条件

//对树高做数学归纳...



BST模板类

```
❖ template <typename T> class BST : public BinTree<T> { //由BinTree派生
public: //以virtual修饰, 以便派生类重写
    virtual BinNodePosi(T) & search( const T & ); //查找
    virtual BinNodePosi(T) insert( const T & ); //插入
    virtual bool remove( const T & ); //删除
protected:
    BinNodePosi(T) _hot; //命中节点的父节点
    BinNodePosi(T) connect34( //3 + 4重构
        BinNodePosi(T), BinNodePosi(T), BinNodePosi(T),
        BinNodePosi(T), BinNodePosi(T), BinNodePosi(T), BinNodePosi(T));
    BinNodePosi(T) rotateAt( BinNodePosi(T) ); //旋转调整
};
```