



搜索/查找 (Search)

- Search基本概念
- Search算法
 - 静态搜索表
 - 二叉搜索树
 - AVL树
- Hashing
- B树
- B+树
- Search算法的分析



二叉搜索树 (Binary Search Tree)

- 是一棵空树，或者是具有下列性质的二叉树：
 - ✓ 每个结点都有一个作为搜索依据的关键码(key)，所有结点的关键码互不相同
 - ✓ 左子树（如果非空）上所有结点的关键码都小于 ($<$) 根结点的关键码
 - ✓ 右子树（如果非空）上所有结点的关键码都大于 ($>$) 根结点的关键码
 - ✓ 左子树和右子树，也是二叉搜索树

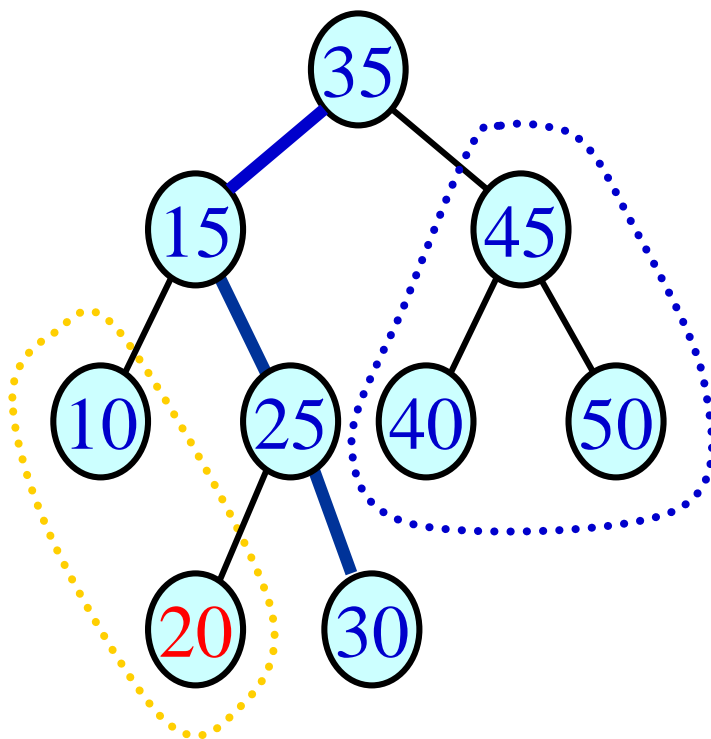
二叉搜索树

- 结点:

- 左子树上所有关键码小于($<$)
结点关键码
- 右子树上所有关键码大于($>$)
结点关键码

- 注意:

若从根结点到某个叶结点有一条路径，路径左边的结点的
关键码 **不一定小于** 路径上的结点的
关键码





二叉排序树(Binary Sorting Tree)

- 如果对一棵二叉搜索树进行中序遍历，可以按从小到大的顺序，将各结点关键码排列起来
- 则这个二叉搜索树，称为二叉排序树



二叉搜索树的类定义

```
#include <iostream.h>
```

```
#include <stdlib.h>
```

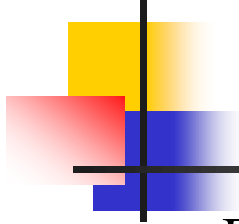
```
template <class E>
```

```
struct BSTNode
```

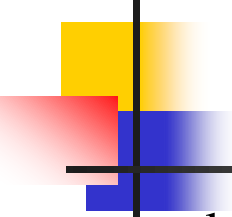
```
{  
    //二叉树结点类
```

```
    E data; //数据域
```

```
    BSTNode<E> *left, *right; //左子女和右子女
```



```
BSTNode() { left = NULL; right = NULL; } //构造函数
BSTNode (const E d, BSTNode<E, K> *L = NULL,
        BSTNode<E, K> *R = NULL)
    { data = d; left = L; right = R;} //构造函数
~BSTNode() {} //析构函数
void setData (E d) { data = d; } //修改
E getData() { return data; } //提取
bool operator < (const E& x) //重载：判小于
    { return data.key < x.key; }
bool operator > (const E& x) //重载：判大于
    { return data.key > x.key; }
bool operator == (const E& x) //重载：判等于
    { return data.key == x.key; }
};
```



```

template <class E>
class BST {          //二叉搜索树类定义
public:
    BST() { root = NULL; }          //构造函数
    BST(EK value);                  //构造函数
    ~BST() {};                      //析构函数
    bool Search (const E x) const
        { return Search(x,root) != NULL; } //搜索
    BST<E>& operator = (const BST<E>& R); //重载：赋值
    void makeEmpty() { makeEmpty (root); root = NULL;} //置空
    void PrintTree() const { PrintTree (root); }          //输出
    E Min() { return Min(root)->data; }          //求最小
    E Max() { return Max(root)->data; }          //求最大
    bool Insert (const E& e1) { return Insert(e1, root); } //插入新元素
    bool Remove (const E x) { return Remove(x, root);} //删除含x的结点

```



```
private:
```

```
    BSTNode<E> *root; //根指针
```

```
    K RefValue;           //输入停止标志
```

```
    BSTNode<E> *
```

```
        Search (const E x, BSTNode<E> *ptr);    //递归：搜索
```

```
void makeEmpty (BSTNode<E> *& ptr);    //递归：置空
```

```
void PrintTree (BSTNode<E> *ptr) const;    //递归：打印
```

```
    BSTNode<E> *
```

```
        Copy (const BSTNode<E> *ptr);           //递归：复
```

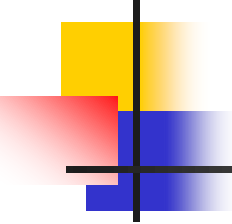
```
    BSTNode<E>* Min (BSTNode<E>* ptr); //递归：求最小
```

```
    BSTNode<E>* Max (BSTNode<E>* ptr); //递归：求最大
```

```
bool Insert (const E& e1, BSTNode<E>* & ptr); //递归：插入
```

```
bool Remove (const E x, BSTNode<E>* & ptr); //递归：删除
```

```
};
```


- 
-
- 二叉搜索树的类定义
 - 用二叉链表作为它的存储表示
 - 许多操作的实现，与二叉树类似



二叉搜索树的搜索算法

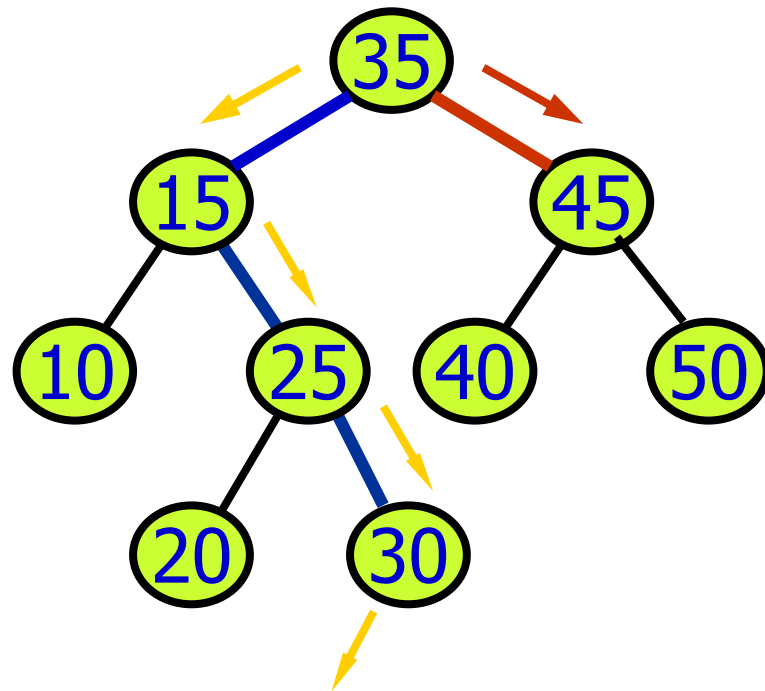
- 在二叉搜索树上进行搜索
 - 一个从根结点开始
 - 沿某一个分支逐层向下
 - 进行比较判等的过程
 - 是一个递归/迭代的过程

搜索28

搜索失败

搜索45

搜索成功



设树的高度为h，最多比较次数不超过h

$$T(n) = O(h)$$



算法描述

- 假设在二叉搜索树中搜索关键码为 x 的元素，搜索过程从根结点开始
- 如果根指针为NULL，则搜索不成功
- 否则用给定值 x 与根结点的关键码进行比较
 - ✓ 若给定值等于根结点关键码，则搜索成功，返回搜索成功信息并报告搜索到结点地址
 - ✓ 若给定值小于根结点的关键码，则继续递归搜索根结点的左子树
 - ✓ 否则，递归搜索根结点的右子树



```
template<class E>
```

```
BSTNode<E,>* BST<E>::
```

```
    Search (const E x, BSTNode<E> *ptr)
```

```
{    //私有递归函数：在以ptr为根的二叉搜索树中搜
```

```
    //索含x的结点。若找到，则函数返回该结点的
```

```
    //地址，否则函数返回NULL值。
```

```
    if (ptr == NULL) return NULL;
```

```
        else if (x < ptr->data) return Search(x, ptr->left);
```

```
        else if (x > ptr->data) return Search(x, ptr->right);
```

```
        else return ptr;                                //搜索成功
```

```
};
```



```
template<class E>
```

```
BSTNode<E>* BST<E>::
```

```
    Search (const K x, BSTNode<E> *ptr)
```

```
{  //非递归函数：作为对比，在当前以ptr为根的二叉搜索树中搜索含x的
```

```
    //结点。若找到，则函数返回该结点的地址，否则函数返回NULL值
```

```
    if (ptr == NULL) return NULL;
```

```
    BSTNode<E>* temp = ptr;
```

```
    while (temp != NULL) {
```

```
        if (x == temp->data) return temp;
```

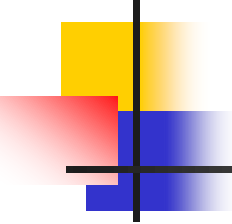
```
        if (x < temp->data) temp = temp->left;
```

```
        else temp = temp->right;
```

```
    }
```

```
    return NULL;
```

```
};
```



■ 搜索过程

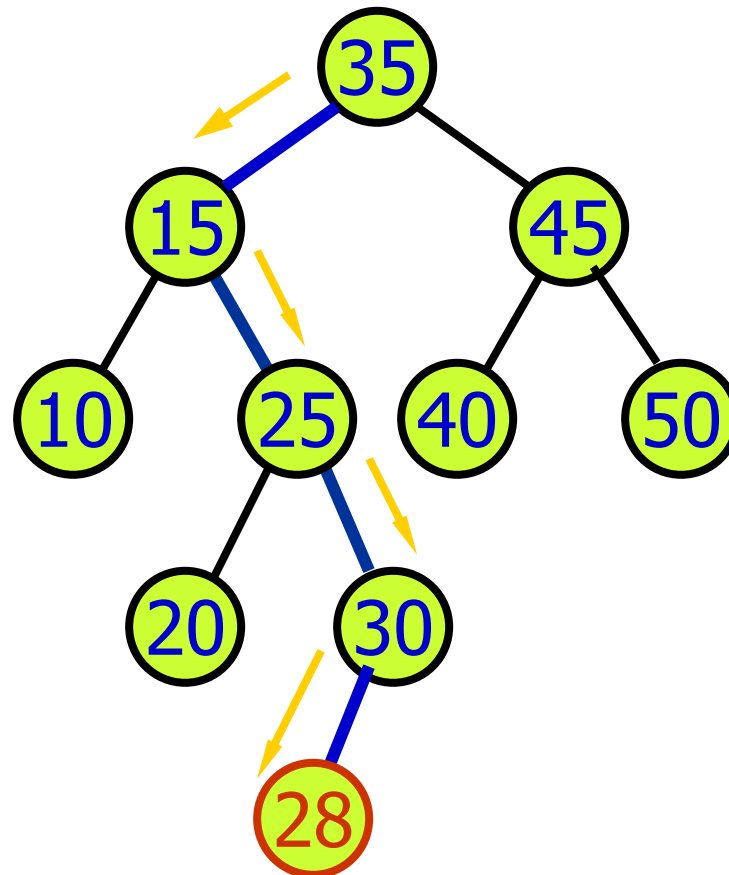
- 从根结点开始，沿某条路径自上而下逐层比较判等的过程
- 搜索成功，搜索指针将停留在树上某个结点
- 搜索不成功，搜索指针将走到树上某个结点的空子树
- 设树的高度为 h ，最多比较次数不超过 h



二叉搜索树的插入算法

- 在二叉搜索树中，插入一个新元素，必须**先检查这个元素是否在树中已经存在**
- 在插入之前，先使用搜索算法在树中检查要插入元素有还是没有
 - 搜索成功，不再插入
 - **搜索不成功**，把新元素加到搜索操作停止的地方
- 每次插入，都要从**根结点**出发搜索插入位置，然后，把新结点作为**叶结点**插入

插入新结点28





二叉搜索树的插入算法

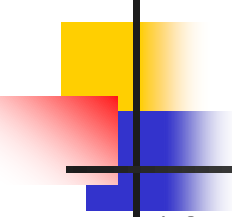
```
template <class E>
```

```
bool BST<E>::Insert (const E& e1, BSTNode<E> *& ptr )
```

```
{
```


```
    //私有函数：在以ptr为根的二叉搜索树中插入值为
```

```
    //e1的结点。若在树中已有含e1的结点则不插入
```

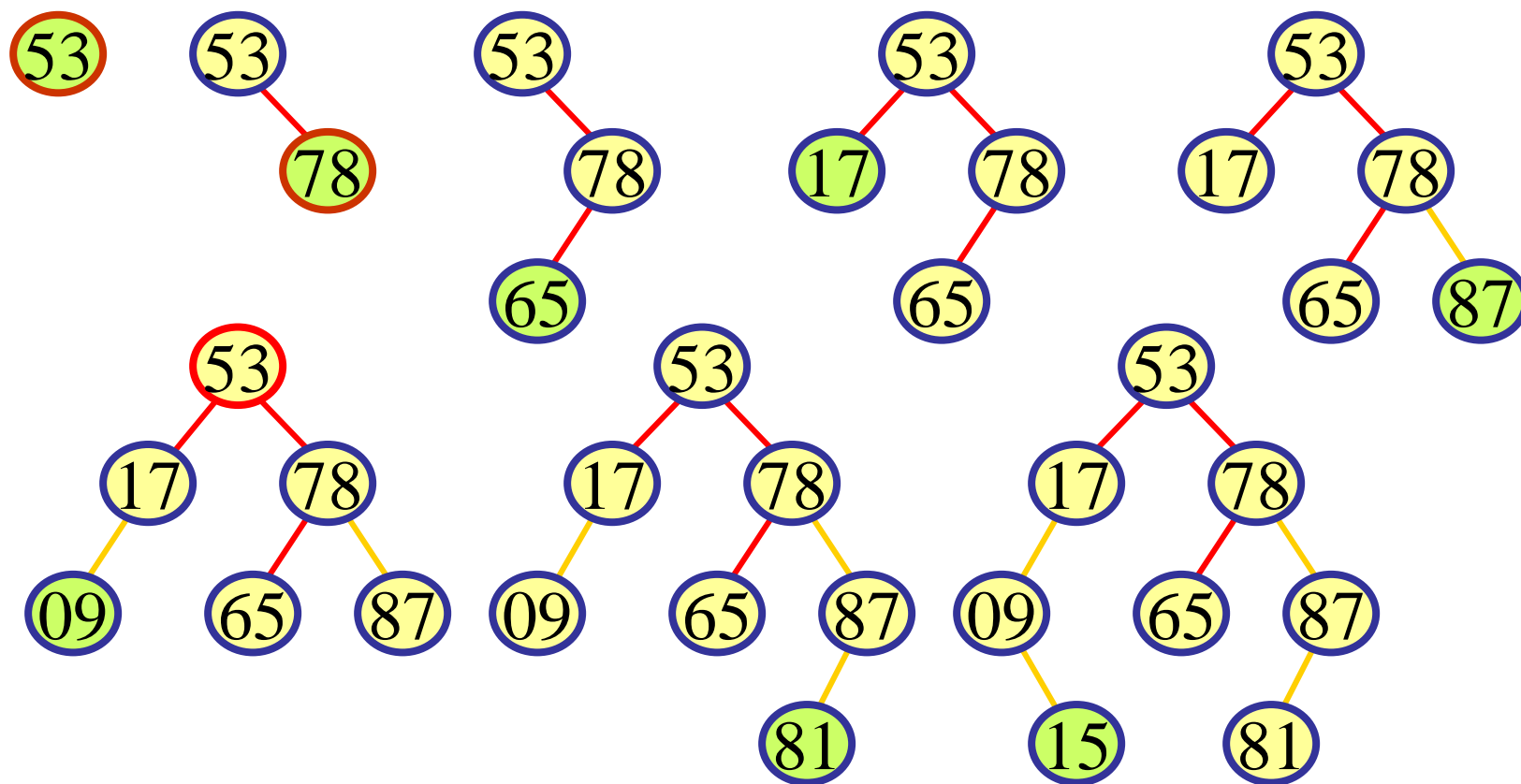


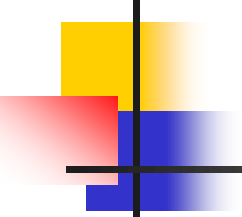
```
if (ptr == NULL) { //新结点作为叶结点插入
    ptr = new BstNode<E>(e1); //创建新结点
    if (ptr == NULL)
        { cerr << "Out of space" << endl; exit(1); }
    return true;
}
else if (e1 < ptr->data) Insert (e1, ptr->left); //左子树插入
else if (e1 > ptr->data) Insert (e1, ptr->right); //右子树插入
else return false; //x已在树中,不再插入
};
```

- 利用二叉搜索树的插入算法， 可以很方便地建立二叉搜索树



输入数据 { 53, 78, 65, 17, 87, 09, 81, 15 }





```
template <class E>
```

```
BST<E>::BST (E value)
```

```
{ //输入一个元素序列, 建立一棵二叉搜索树
```

```
    E x;
```

```
    root = NULL; RefValue = value;    //置空树
```

```
    cin >> x;                        //输入数据
```

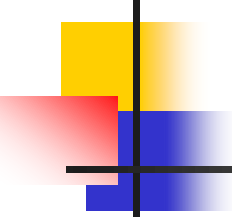
```
    while ( x.key != RefValue) {
```

```
        //RefValue是一个输入结束标志
```

```
        Insert (x, root); cin >> x; //插入, 再输入数据
```

```
    }
```

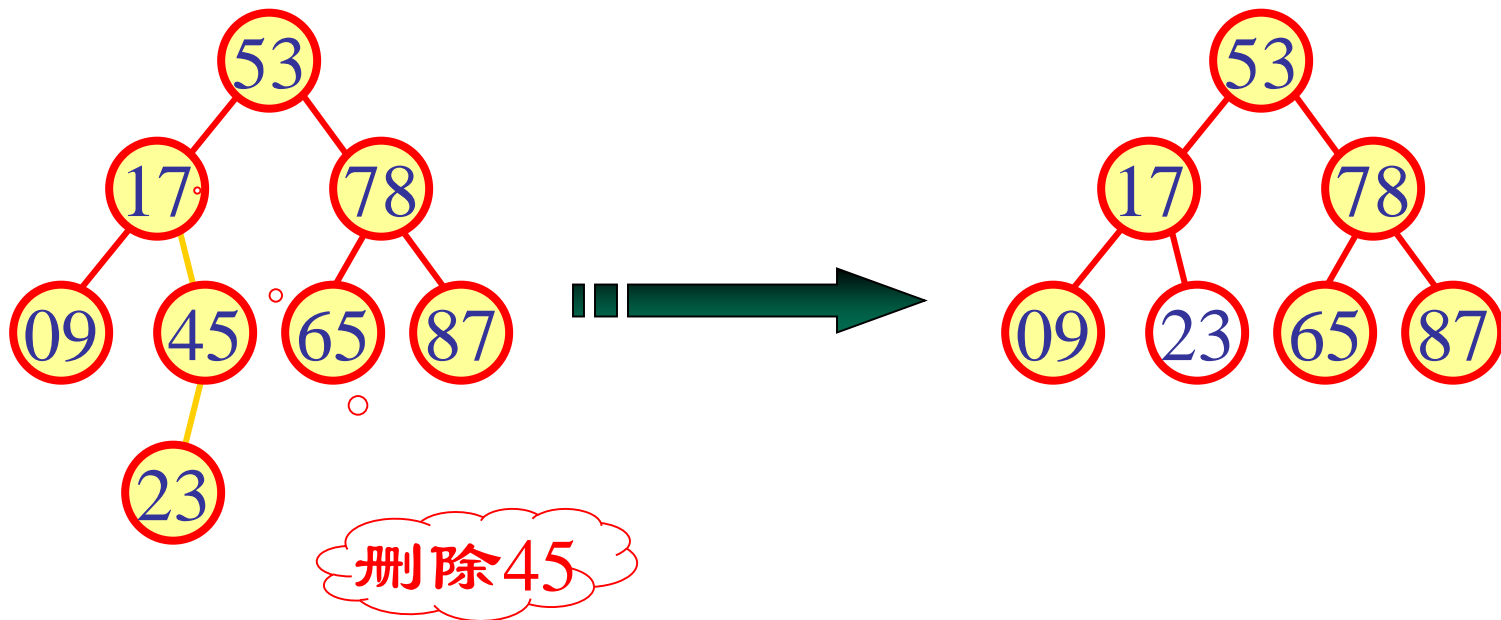
```
};
```



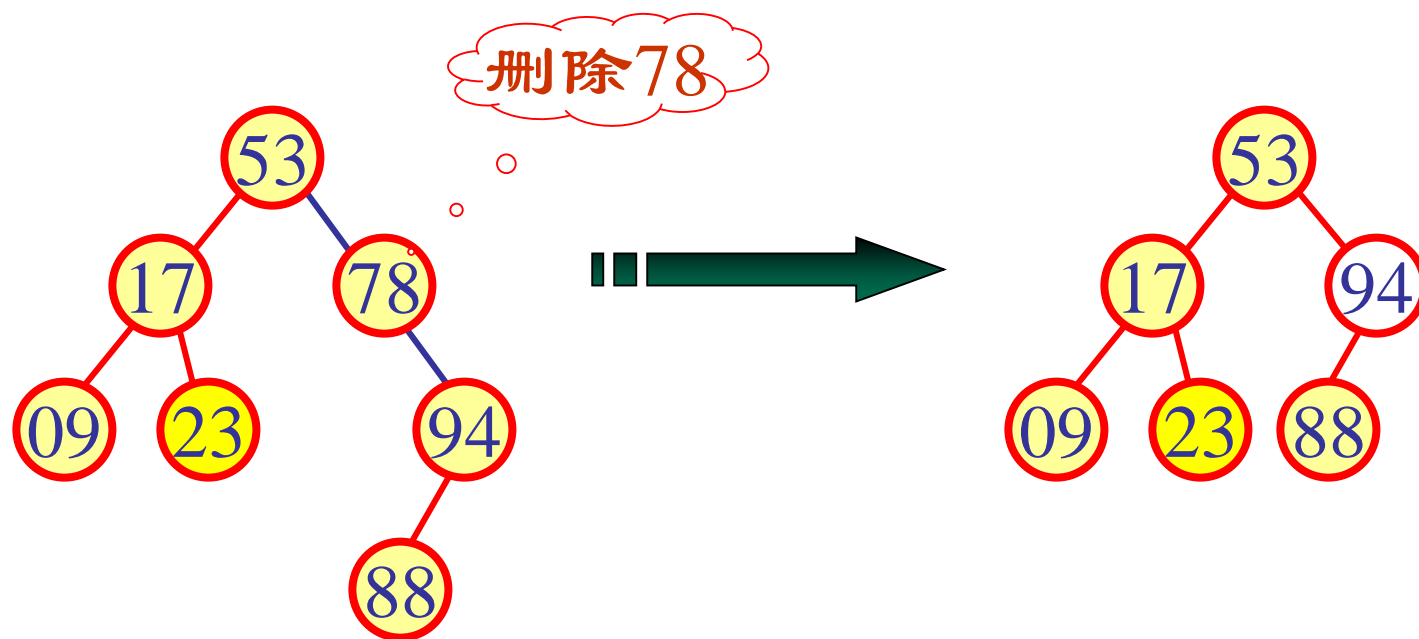
二叉搜索树的删除算法

- 必须将因删除结点而断开的二叉链表重新链接起来
- 同时，确保二叉搜索树的性质不会失去
- 为保证在删除后树的搜索性能不至于降低，还需要防止重新链接后树的高度增加
 - ✓ 删除叶结点：将其双亲结点指向它的指针清零，再释放它。
 - ✓ 被删结点右子树为空：其左子女结点顶替它的位置，再释放它。
 - ✓ 被删结点左子树为空：其右子女结点顶替它的位置，再释放它。
 - ✓ 被删结点左、右子树都不为空
 - ✓ 其右子树中寻找中序下的第一个结点(关键码最小), 用该节点的值填补到被删结点中，再来处理该结点的删除问题。

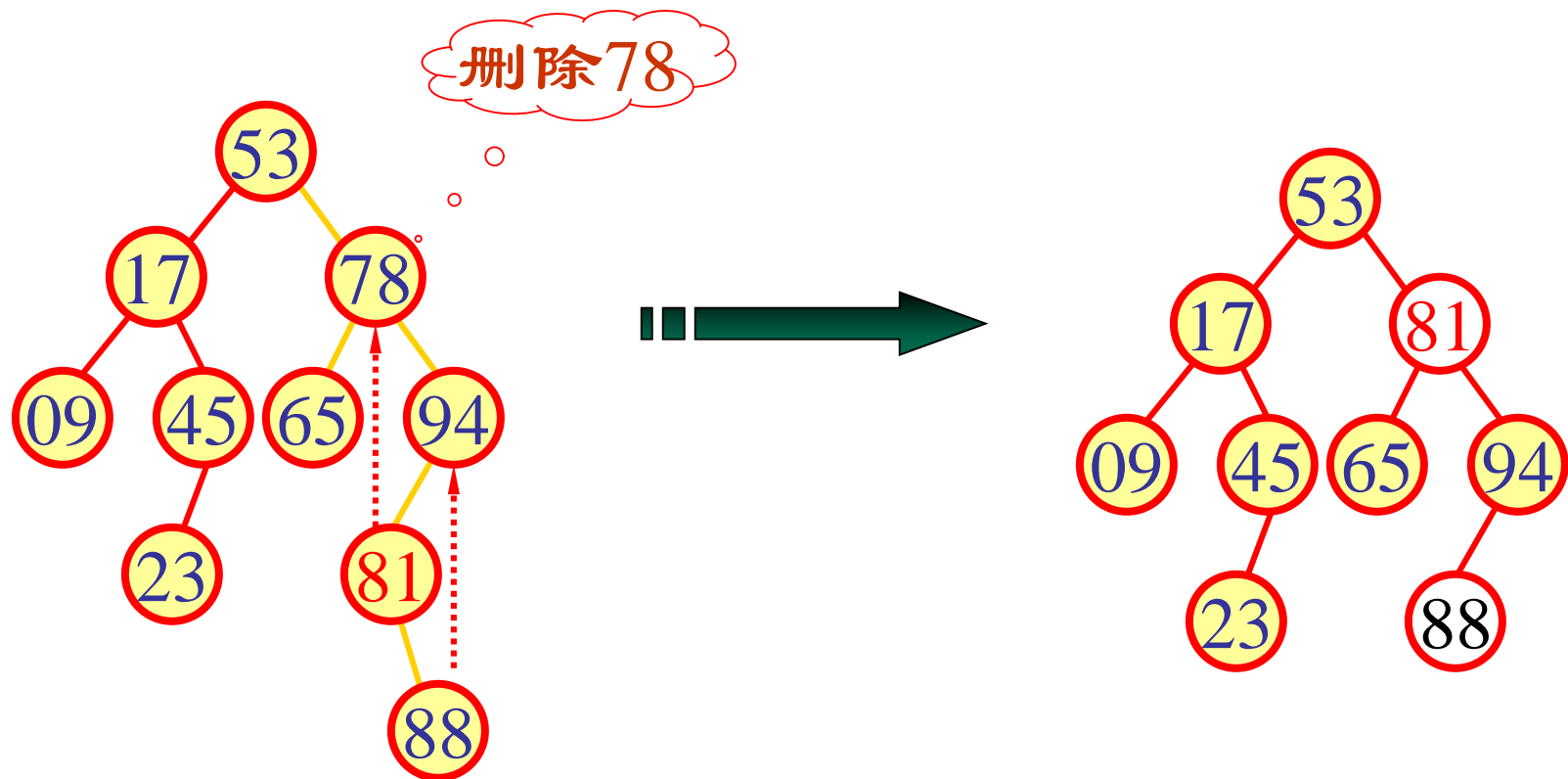
右子树空, 用左子女顶替

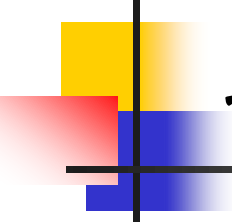


左子树空, 用右子女顶替



在**右子树**上找中序下第一个结点**填补**





二叉搜索树的删除算法

```
template <class E >
bool BST<E>::Remove (const E x, BstNode<E> *& ptr)
{ //在以 ptr 为根的二叉搜索树中删除含 x 的结点
    BstNode<E> *temp;
    if (ptr != NULL)
    {
        if (x < ptr->data) Remove (x, ptr->left);
                                //在左子树中执行删除
        else if (x > ptr->data) Remove (x, ptr->right);
                                //在右子树中执行删除
    }
}
```



```
else if (ptr->left != NULL && ptr->right != NULL)
```

```
{    //ptr指示关键码为x的结点， 它有两个子女
```

```
    temp = ptr->right;
```

```
    //到右子树搜寻中序下第一个结点
```

```
while (temp->left != NULL)
```

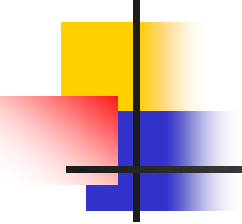
```
    temp = temp->left;
```

```
    ptr->data = temp->data;
```

```
    //用该结点数据代替根结点数据
```

```
Remove (ptr->data, ptr->right);
```

```
}
```

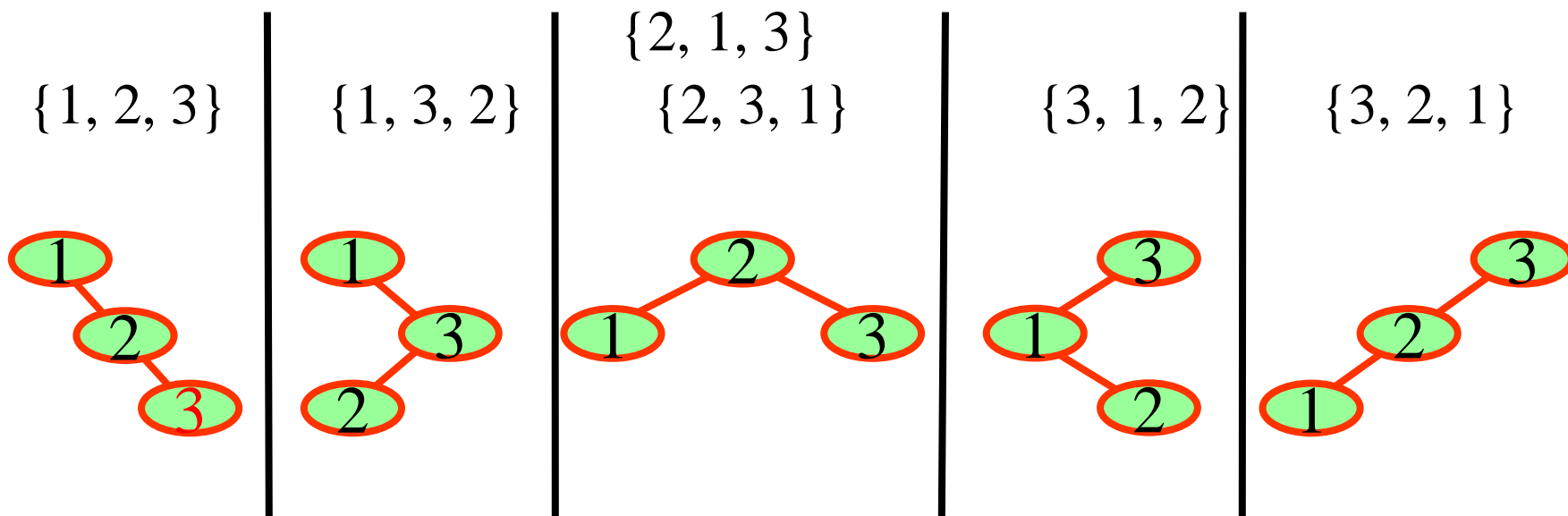


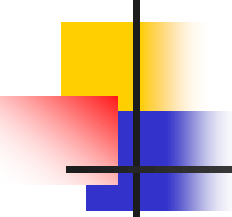
```
else { //ptr指示关键码为x的结点有一个子女
    temp = ptr;
    if (ptr->left == NULL) ptr = ptr->right;
    else ptr = ptr->left;
    delete temp;
    return true;
}
return false;
};
```

- 注意：在删除算法参数表引用型指针参数ptr的使用

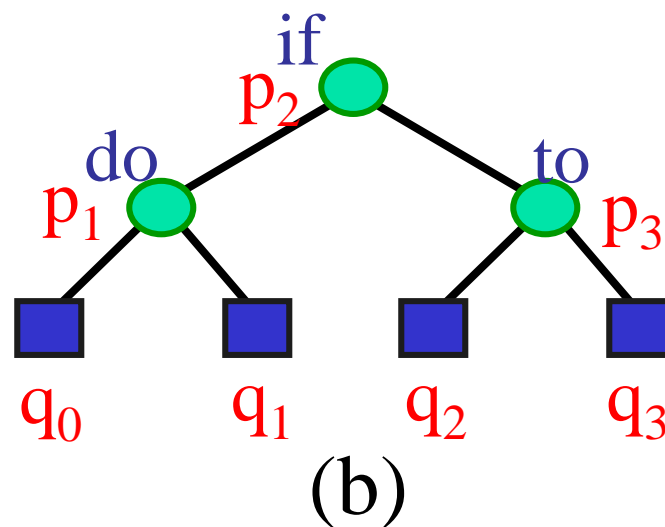
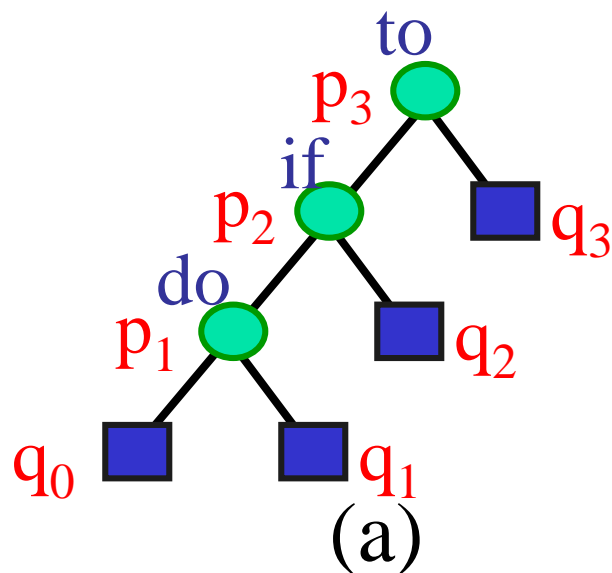
二叉搜索树性能分析

- 对于有 n 个关键码的集合，其关键码有 $n!$ 种不同排列，可构成不同二叉搜索树有 $\frac{1}{n+1}C_{2n}^n$ (棵)

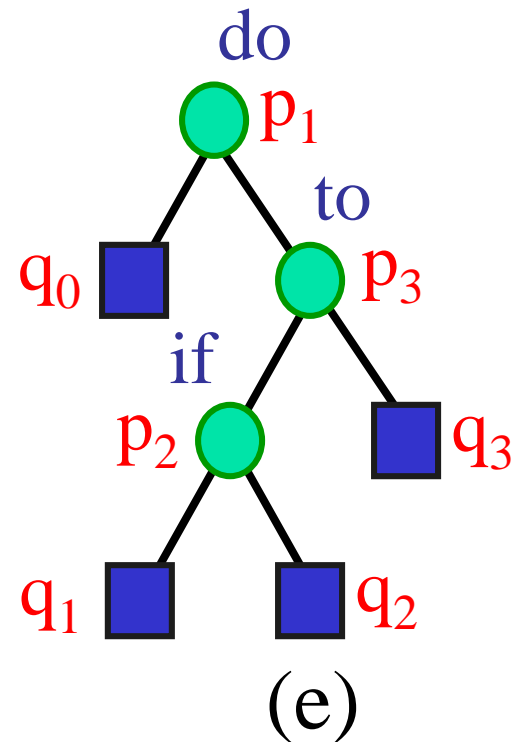
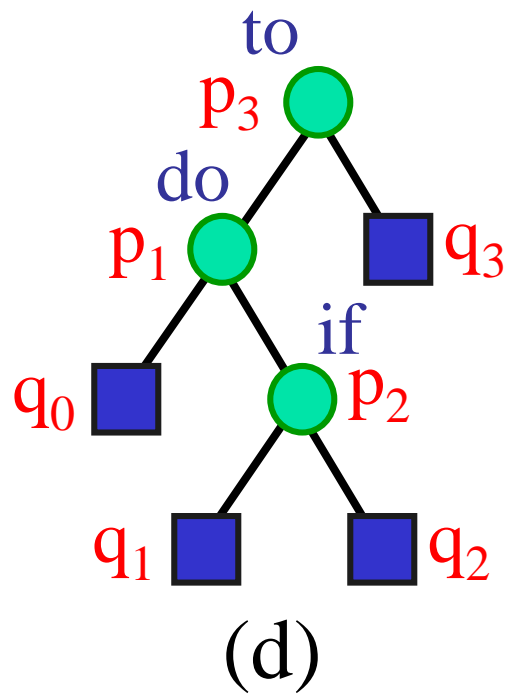
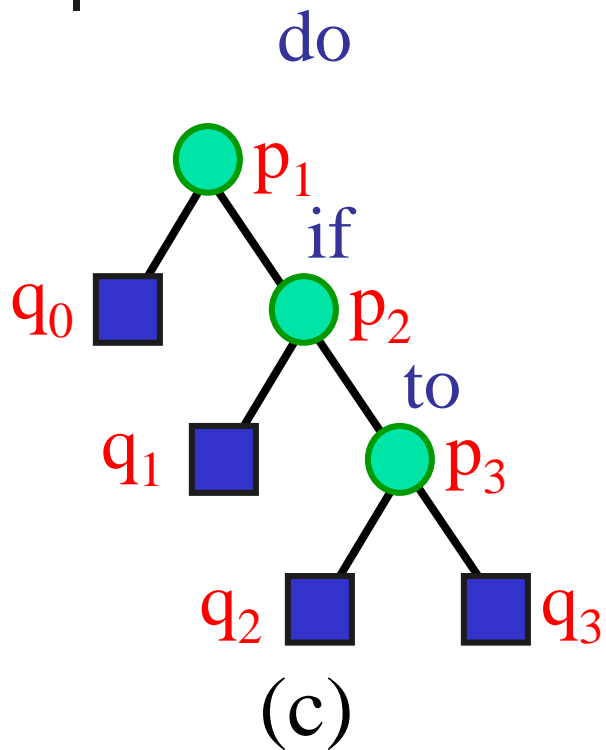


- 
- 3 个数据{ 1, 2, 3 }, 输入顺序不同, 建立起来的二叉搜索树的形态不同, 则二叉搜索树的搜索性能也不同
 - 如果输入序列选得不好, 会建立起一棵单支树, 使得二叉搜索树的高度达到最大
 - 如何评价这些二叉搜索树搜索效率?
 - 在二叉搜索树中加入外结点, 形成判定树
 - 外结点表示失败结点, 内结点表示搜索树中已有的数据
 - 这样的判定树即为扩充的二叉搜索树

- 已知关键码集合 $\{a_1, a_2, a_3\} = \{\text{do}, \text{if}, \text{to}\}$
- 对应搜索概率 p_1, p_2, p_3 ,
- 搜索不成功的搜索概率分别为 q_0, q_1, q_2, q_3
- 可能的二叉搜索树如下所示



判定树





■ 在判定树中

- ◆ ○表示内部结点

- ◆ 包含了关键码集合中的某一个关键码

- ◆ □表示外部结点

- ◆ 代表各关键码间隔中的不在关键码集合中的关键码

- 在每两个外部结点间必存在一个内部结点

- 
- 搜索成功的平均搜索长度 ASL_{succ}

$$ASL_{succ} = \sum_{i=1}^n p[i] * l[i].$$

- 所有内部结点上的搜索概率 $p[i]$
 - 搜索该结点时所需的关键码比较次数 $c[i]$ ($= l[i]$, 即结点所在层次)
 - 乘积之和
- 设各关键码的搜索概率相等: $p[i] = 1/n$

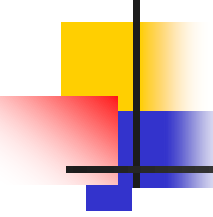
$$ASL_{succ} = \frac{1}{n} \sum_{i=1}^n l[i].$$

- 
- 搜索不成功，平均搜索长度 ASL_{unsucc}

$$ASL_{unsucc} = \sum_{j=0}^n q[j] * (l'[j] - 1).$$

- 所有外部结点上搜索概率 $q[j]$
 - 到达外部结点所需关键码比较次数 $c'[j](= l'[j])$
 - 乘积之和
- 设外部结点搜索概率相等: $q[j] = 1/(n+1)$

$$ASL_{unsucc} = \frac{1}{n+1} \sum_{j=0}^n (l'[j] - 1).$$



(1) 相等搜索概率的情形

- 设树中所有内、外部结点的搜索概率都相等:

$$p[i] = 1/3, 1 \leq i \leq 3, q[j] = 1/4, 0 \leq j \leq 3$$

图(a): $ASL_{succ} = 1/3 * 3 + 1/3 * 2 + 1/3 * 1 = 6/3$

$$ASL_{unsucc} = 1/4 * 3 * 2 + 1/4 * 2 + 1/4 * 1 = 9/4$$

图(b): $ASL_{succ} = 1/3 * 2 * 2 + 1/3 * 1 = 5/3$

$$ASL_{unsucc} = 1/4 * 2 * 4 = 8/4$$

图(c): $ASL_{succ} = 1/3 * 1 + 1/3 * 2 + 1/3 * 3 = 6/3$

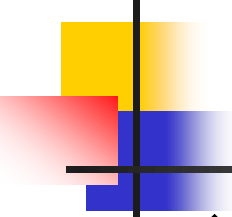
$$ASL_{unsucc} = 1/4 * 1 + 1/4 * 2 + 1/4 * 3 * 2 = 9/4$$

图(d): $ASL_{succ} = 1/3 * 2 + 1/3 * 3 + 1/3 * 1 = 6/3$

$$ASL_{unsucc} = 1/4 * 2 + 1/4 * 3 * 2 + 1/4 * 1 = 9/4$$

图(e): $ASL_{succ} = 1/3 * 1 + 1/3 * 3 + 1/3 * 2 = 6/3$

$$ASL_{unsucc} = 1/4 * 1 + 1/4 * 3 * 2 + 1/4 * 2 = 9/4$$

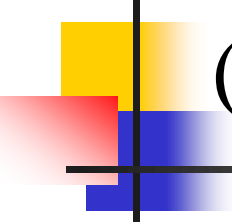
- 
- 平均搜索长度达到最小的扩充的二叉搜索树，称作最优二叉搜索树
 - 在相等搜索概率的情形下，所有内部、外部结点的搜索概率都相等，视它们的权值都为 1
 - 第 k 层有 2^{k-1} 个结点， $k = 1, 2, \dots$
 - 则有 n 个内部结点的扩充二叉搜索树的内部路径长度 I 至少等于下面序列的前 n 项的和

0, 1, 1, 2, 2, 2, 2, 3, 3, 3, 3, 3, 3, 3, 3, 4, 4, ...

因此，最优二叉搜索树的搜索成功的平均搜索长度和搜索不成功的平均搜索长度分别为：

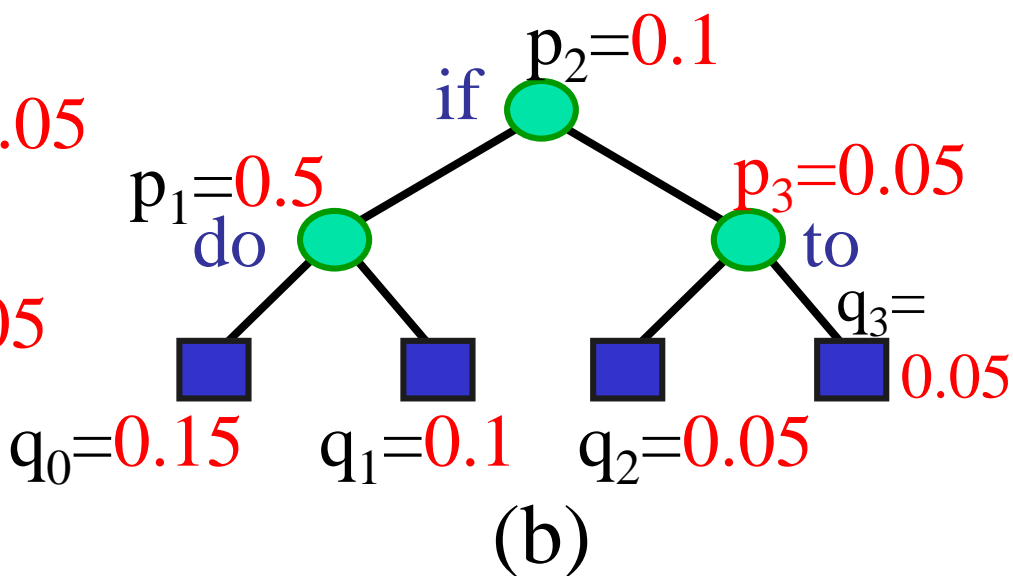
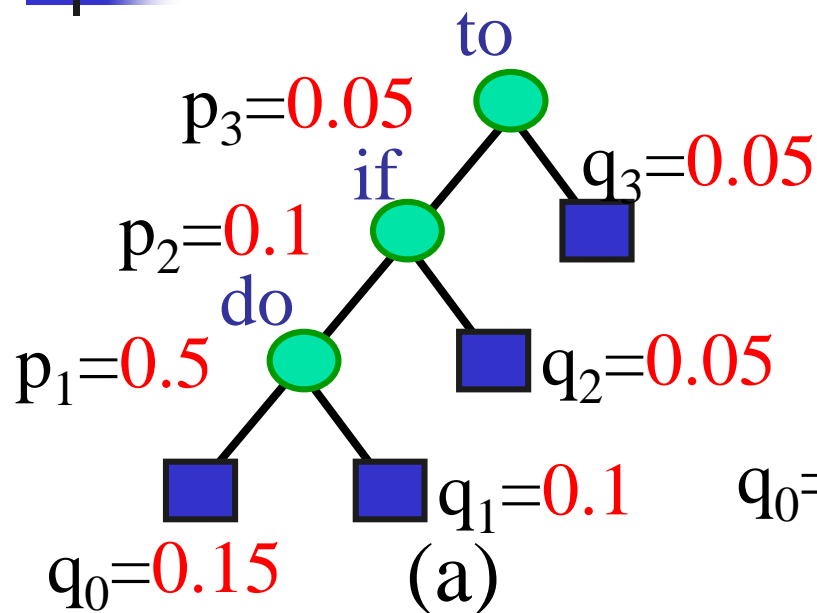
$$ASL_{succ} = \sum_{i=1}^n (\lfloor \log_2 i \rfloor + 1).$$

$$ASL_{unsucc} = \sum_{i=n+1}^{2n+1} (\lfloor \log_2 i \rfloor).$$



(2) 不相等搜索概率的情形

- 设二叉搜索树中所有内、外部结点的搜索概率互不相等
$$p[1] = 0.5, p[2] = 0.1, p[3] = 0.05$$
$$q[0] = 0.15, q[1] = 0.1, q[2] = 0.05, q[3] = 0.05$$
- 分别计算
 - 各个可能的扩充二叉搜索树的搜索性能
- 判断
 - 哪些扩充二叉搜索树的平均搜索长度最小

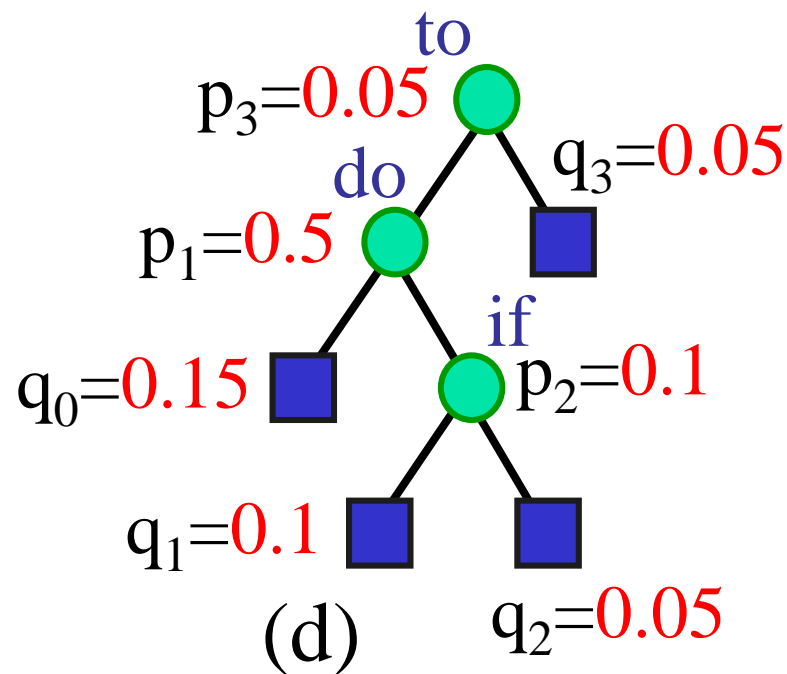
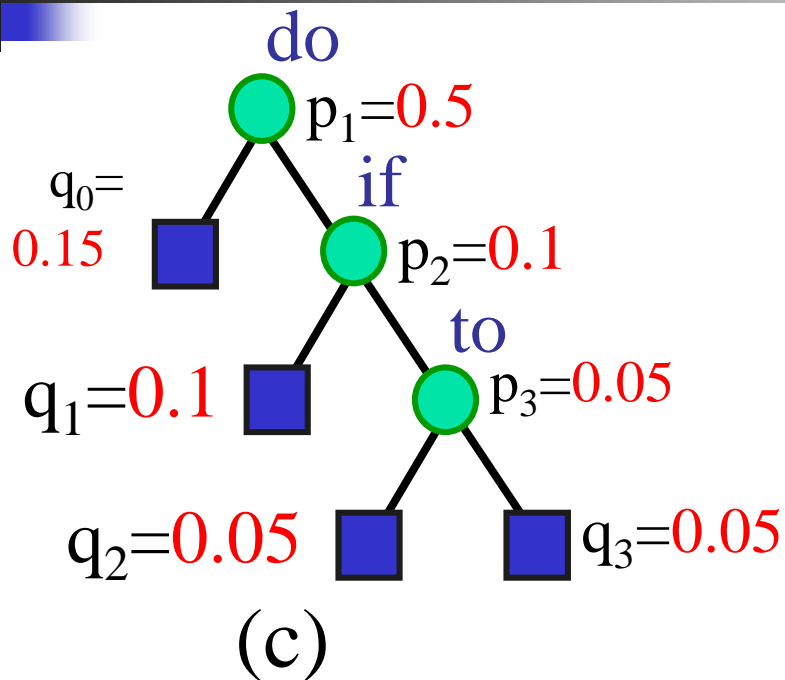


图(a): $ASL_{succ} = 0.5 * 3 + 0.1 * 2 + 0.05 * 1 = 1.75$

$ASL_{unsucc} = 0.15 * 3 + 0.1 * 3 + 0.05 * 2 + 0.05 * 1 = 0.9$

图(b): $ASL_{succ} = 0.5 * 2 + 0.1 * 1 + 0.05 * 2 = 1.2$

$ASL_{unsucc} = (0.15 + 0.1 + 0.05 + 0.05) * 2 = 0.7$

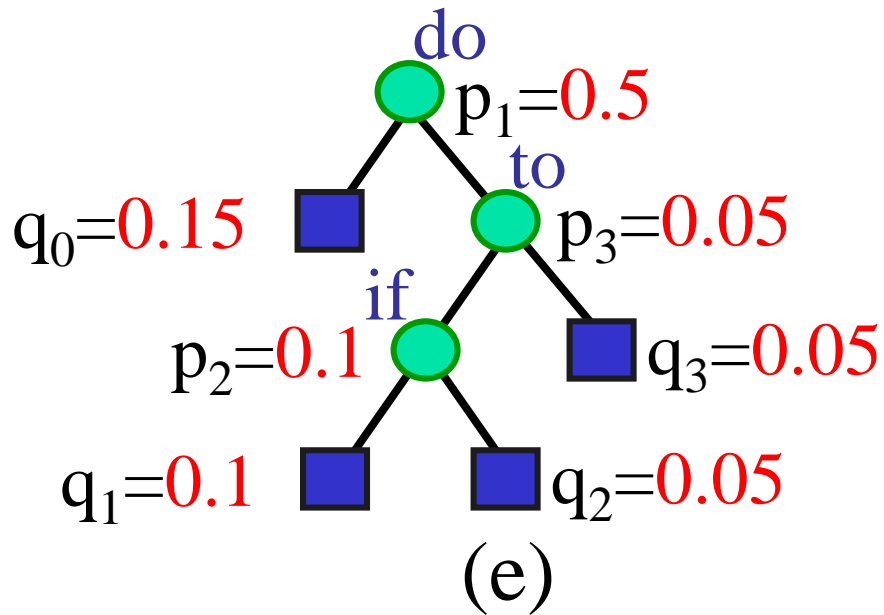


图(c): $ASL_{succ} = 0.5*1+0.1*2+0.05*3 = 0.85$

$$ASL_{unsucc} = 0.15*1+0.1*2+0.05*3+0.05*3 = 0.75$$

图(d) : $ASL_{succ} = 0.5*2+0.1*3+0.05*1 = 1.35$

$$ASL_{unsucc} = 0.15*2+0.1*3+0.05*3+0.05*1 = 0.8$$



图(e) :

$$ASL_{succ} = 0.5 * 1 + 0.1 * 3 + 0.05 * 2 = 0.9$$

$$ASL_{unsucc} = 0.15 * 1 + 0.1 * 3 + 0.05 * 3 + 0.05 * 2 = 0.7$$

- 图(c)和图(e)的情形下树的平均搜索长度达到最小
- 图(c)和图(e)的情形是最优二叉搜索树



最优二叉搜索树

- 在相等搜索概率的情形下
 - 所有内、外部结点的搜索概率都相等，它们的权值都视为1
 - 有 n 个内部结点的最优二叉搜索树应为
 - 完全二叉树或理想平衡树
- 不相等搜索概率情形下
 - 最优二叉搜索树可能不同于完全二叉树的形态