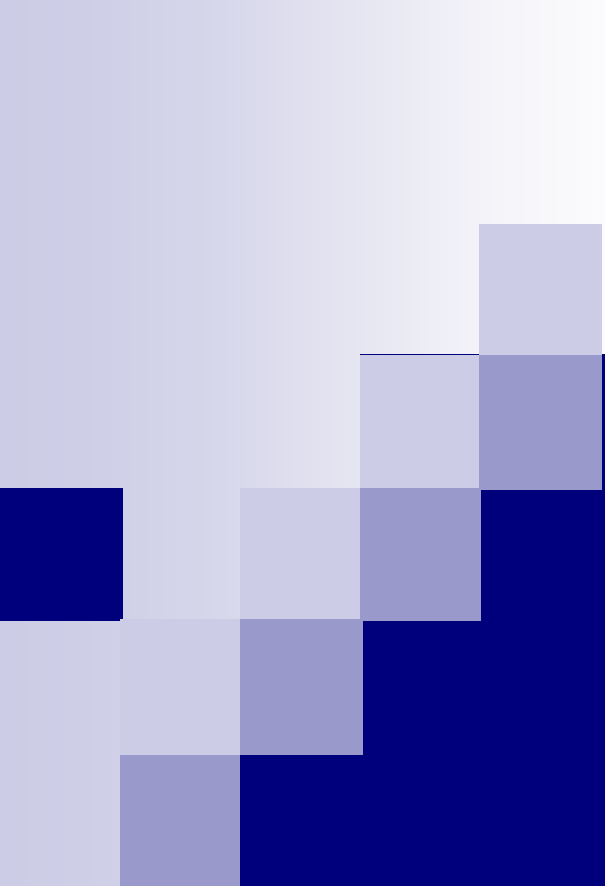




■ 上节重点回顾

- 哈夫曼树（最优二叉树）
- 哈夫曼编码
- WPL



第七章 搜索结构

本章的主要内容是:

- 搜索的基本概念
- 线性表的搜索技术
- 树表的搜索技术
- 散列表的搜索技术

搜索(Search)的概念

- 所谓搜索，就是在数据集合中寻找满足某种条件的数据对象。
- 搜索的结果通常有两种可能：
 - 搜索成功
 - 搜索不成功

7.1 概述

基本概念

- **关键码**：可以标识一个记录的某个数据项。
- **键值**：关键码的值。
- **主关键码**：可以唯一地标识一个记录的关键码。
- **次关键码**：不能唯一地标识一个记录的关键码。

职工号	姓名	性别	年龄	工作时间
0001	王刚	男	38	1990.4
0002	张亮	男	25	2003.7
0003	刘楠	女	47	1979.9
0004	齐梅	女	25	2003.7
0005	李爽	女	50	1972.9

7.1 概述

查找的基本概念

- **静态搜索**：不涉及插入和删除操作的搜索。
- **动态搜索**：涉及插入和删除操作的搜索。

静态搜索适用于：搜索集合一经生成，便只对其进行搜索，而不进行插入和删除操作，或经过一段时间的搜索之后，集中地进行插入和删除等修改操作；

动态搜索适用于：查找与插入和删除操作在同一个阶段进行，例如当查找成功时，要删除查找到的记录，当查找不成功时，要插入被查找的记录。

7.1 概述

基本概念

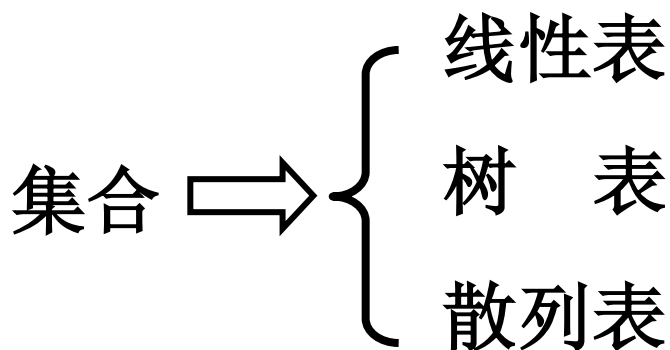
搜索结构：面向搜索操作的数据结构，即搜索基于的数据结构。

查找结构



查找方法

集合中元素之间不存在明显的组织规律，不便查找。



7.1 概述

基本概念

本章讨论的搜索结构：

- **线性表**：适用于静态搜索，主要采用顺序搜索技术和折半搜索技术。
- **树表**：适用于动态搜索，主要采用二叉排序树的搜索技术。
- **散列表**：静态查找和动态查找均适用，主要采用散列技术。

7.1 概述

搜索算法的性能

搜索算法时间性能通过关键码的比较次数来度量。

① 关键码的比较次数与哪些因素有关呢？

平均搜索长度：将搜索算法进行的关键码的比较次数的数学期望值定义为**平均搜索长度**，即：

$$ASL = \sum_{i=1}^n p_i c_i$$

其中： n ：问题规模，查找集合中的记录个数；

p_i ：搜索第 i 个记录的概率；

c_i ：搜索第 i 个记录所需的关键码的比较次数。

7.1 概述

搜索算法的性能

$$ASL = \sum_{i=1}^n p_i c_i$$

c_i 取决于算法； p_i 与算法无关，取决于具体应用。如果 p_i 是已知的，则平均查找长度只是问题规模的函数。

7.1.1 静态搜索表

- 在静态搜索表中，数据元素存放于数组中，利用数组元素的下标作为数据元素的存放地址。搜索算法根据给定值 k ，在数组中进行搜索。直到找到 k 在数组中的存放位置或可确定在数组中找不到 k 为止。

数据表与搜索表的类定义

```
#include <iostream.h>
#include <assert.h>
const int defaultSize = 100;
template <class E, class K>
class dataList;           //数据表类的前视定义

template <class E, class K >
class dataNode {           //数据表中结点类的定义
friend class dataList<E, K>;
private:
    K key;                 //关键码域
    E other;               //其他域 (视问题而定)
                           //声明其友元类为dataList
public:
```

```
dataNode (const K x) : key(x) { }           //构造函数  
K getKey() const { return key; }           //读取关键码  
void setKey (K x) { key = x; }             //修改关键码  
};
```

```
template <class E, class K >  
class dataList {                             //数据表类定义  
protected:  
    dataNode<E, K> *Element;                 //数据表存储数组  
    int ArraySize, CurrentSize;  
                                             //数组最大长度和当前长度
```

7.1.2 顺序搜索 (Sequential Search)

- 顺序搜索主要用于在线性表中搜索。
 - 顺序用各元素的关键码与给定值 x 进行比较
 - **技巧**：把待查关键字`key`存入表头或表尾（俗称“哨兵”），这样可以加快执行速度。

顺序查找（线性查找）

基本思想：从线性表的一端向另一端逐个将关键码与给定值进行比较，若相等，则查找成功，给出该记录在表中的位置；若整个表检测完仍未找到与给定值相等的关键码，则查找失败，给出失败信息。

例：查找 $k=35$

0	1	2	3	4	5	6	7	8	9
	10	15	24	6	12	35	40	98	55
						\uparrow_i	\uparrow_i	\uparrow_i	\uparrow_i

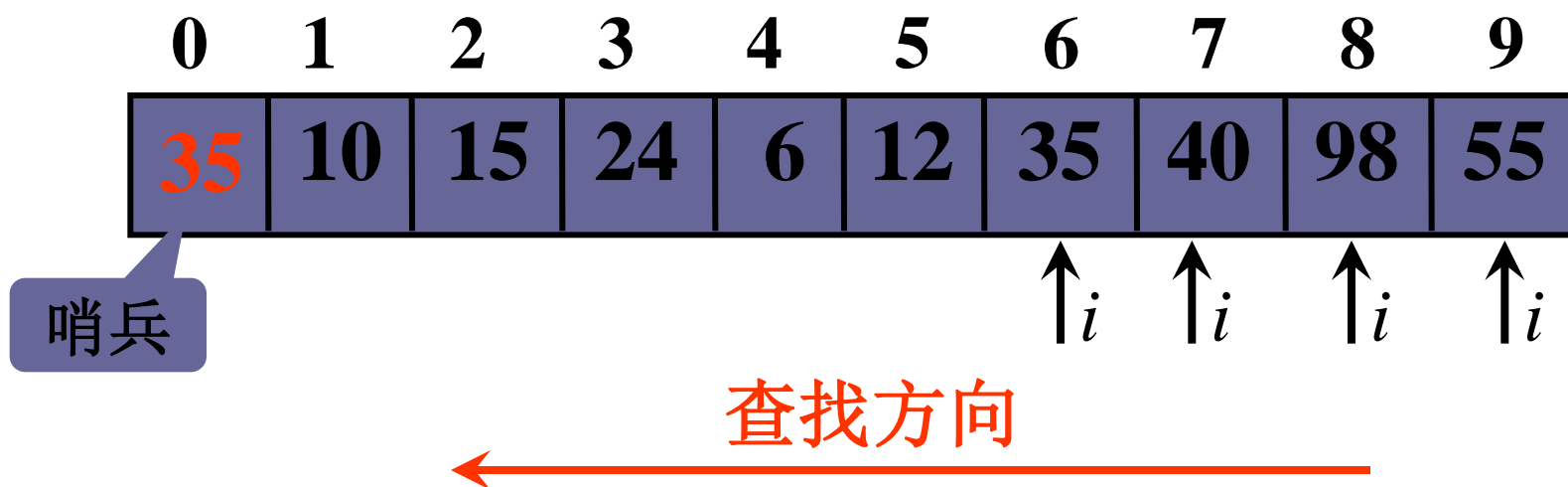
顺序查找（线性查找）

```
int SeqSearch1 (int r[ ], int n, int k)  
//数组r[1] ~ r[n]存放查找集合  
{  
    i = n;  
    while (i>0 && r[i] != k)  
        i--;  
    return i;  
}
```


改进的顺序查找

基本思想：设置“哨兵”。哨兵就是待查值，将它放在查找方向的**尽头**处，免去了在查找过程中每一次比较后都要判断查找位置是否**越界**，从而提高查找速度

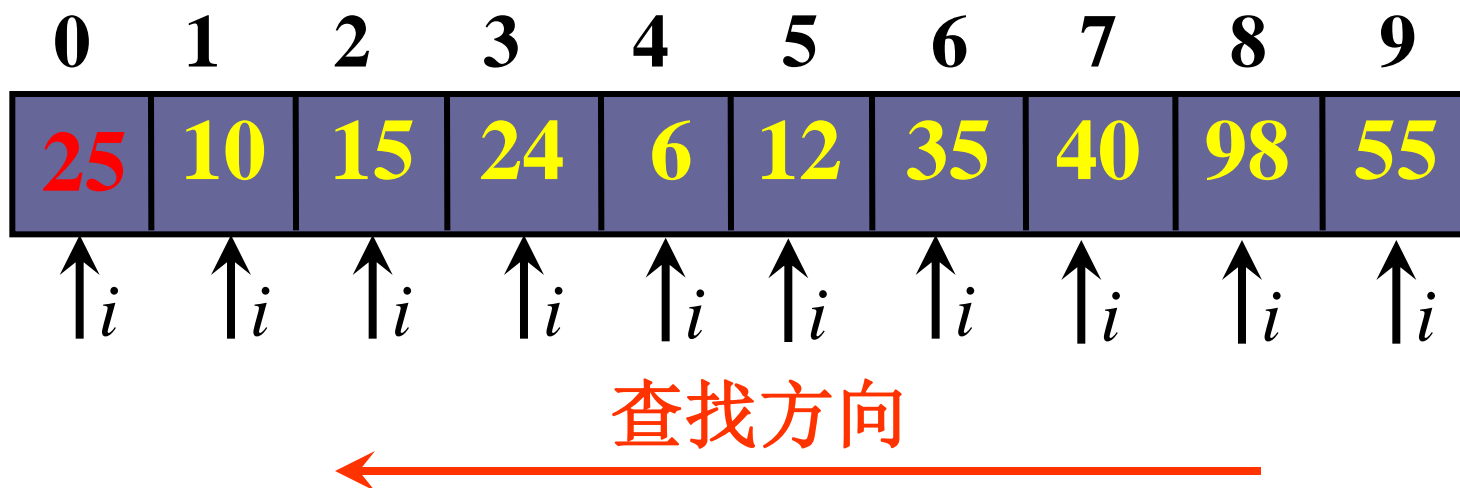
例：查找 $k=35$



改进的顺序查找

基本思想：设置“哨兵”。哨兵就是待查值，将它放在查找方向的尽头处，免去了在查找过程中每一次比较后都要判断查找位置是否越界，从而提高查找速度

例：查找 $k=25$



改进的顺序查找

```
int SeqSearch2(int r[ ], int n, int k)
//数组r[1] ~ r[n]存放查找集合
{
    r[0] = k; i = n;
    while (r[i] != k)
        i--;
    return i;
}
```

$$ASL = \sum_{i=1}^n p_i c_i = \sum_{i=1}^n p_i (n - i + 1) = (n+1)/2 = O(n)$$

顺序查找的缺点：

平均查找长度较大，特别是当待查找集合中元素较多时，查找效率较低。

顺序查找的优点： 算法简单而且使用面广。

- 对表中记录的存储没有任何要求，顺序存储和链接存储均可；
- 对表中记录的有序性也没有要求，无论记录是否按关键码有序均可。

7.1.3 基于有序顺序表的折半搜索

- 哨兵法当N很大时，搜索的效率很低。
- 如果把顺序表的元素按其关键码从小到大排列，则可以采取效率更高的搜索算法。

折半搜索

使用条件:

- 线性表中的记录必须按关键码有序;
- 必须采用顺序存储。

基本思想: 折半搜索时, 先求位于搜索区间正中的对象的下标 mid , 用其关键码与给定值 x 比较:

$Element[mid].key == x$, 搜索成功;

$Element[mid].key > x$, 把搜索区间缩小到表的前半部分, 继续折半搜索;

$Element[mid].key < x$, 把搜索区间缩小到表的后半部分, 继续折半搜索。

例：查找值为14的记录的过程：

0 1 2 3 4 5 6 7 8 9 10 11 12 13

	7	14	18	21	23	29	31	35	38	42	46	49	52
--	---	----	----	----	----	----	----	----	----	----	----	----	----

↑
low=1



18>14



↑
mid=7

31>14



high=13

↑
mid=3

high=6

↑
high=2



mid=1

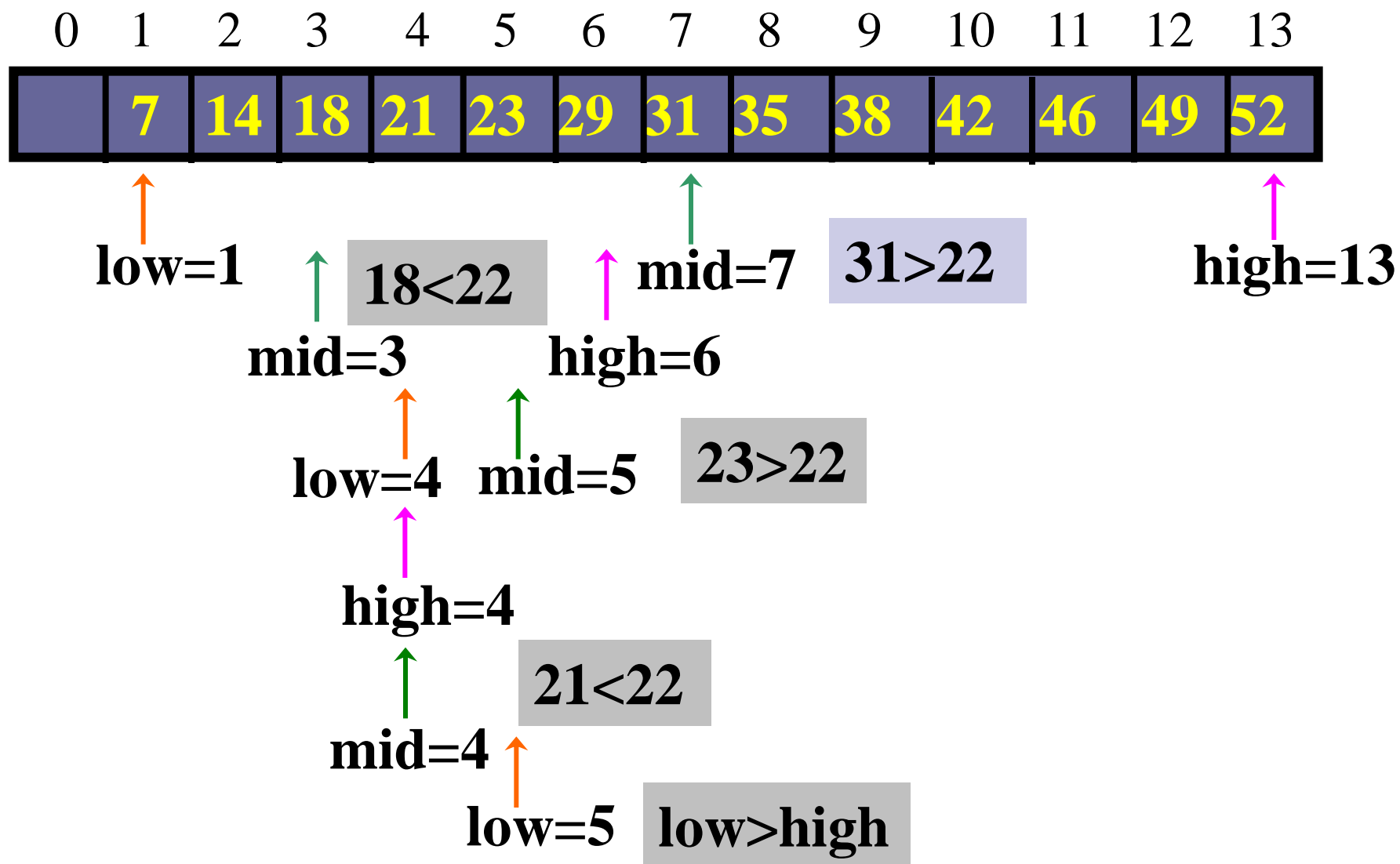
7<14

↑
low=2

↑
mid=2

14=14

例：查找值为22的记录的过程：



折半查找——非递归算法

```
int BinSearch1(int r[ ], int n, int k)
{
    //数组r[1] ~ r[n]存放查找集合
    low = 1; high = n;
    while (low <= high)
    {
        mid = (low + high) / 2;
        if (k < r[mid]) high = mid - 1;
        else if (k > r[mid]) low = mid + 1;
        else return mid;
    }
    return 0;
}
```

折半查找——递归算法

```
int BinSearch2(int r[ ], int low, int high, int k)
{
    //数组r[1] ~ r[n]存放查找集合
    if (low > high) return 0;
    else {
        mid = (low + high) / 2;
        if (k < r[mid])
            return BinSearch2(r, low, mid-1, k);
        else if (k > r[mid])
            return BinSearch2(r, mid+1, high, k);
        else return mid;
    }
}
```

折半查找——递归算法

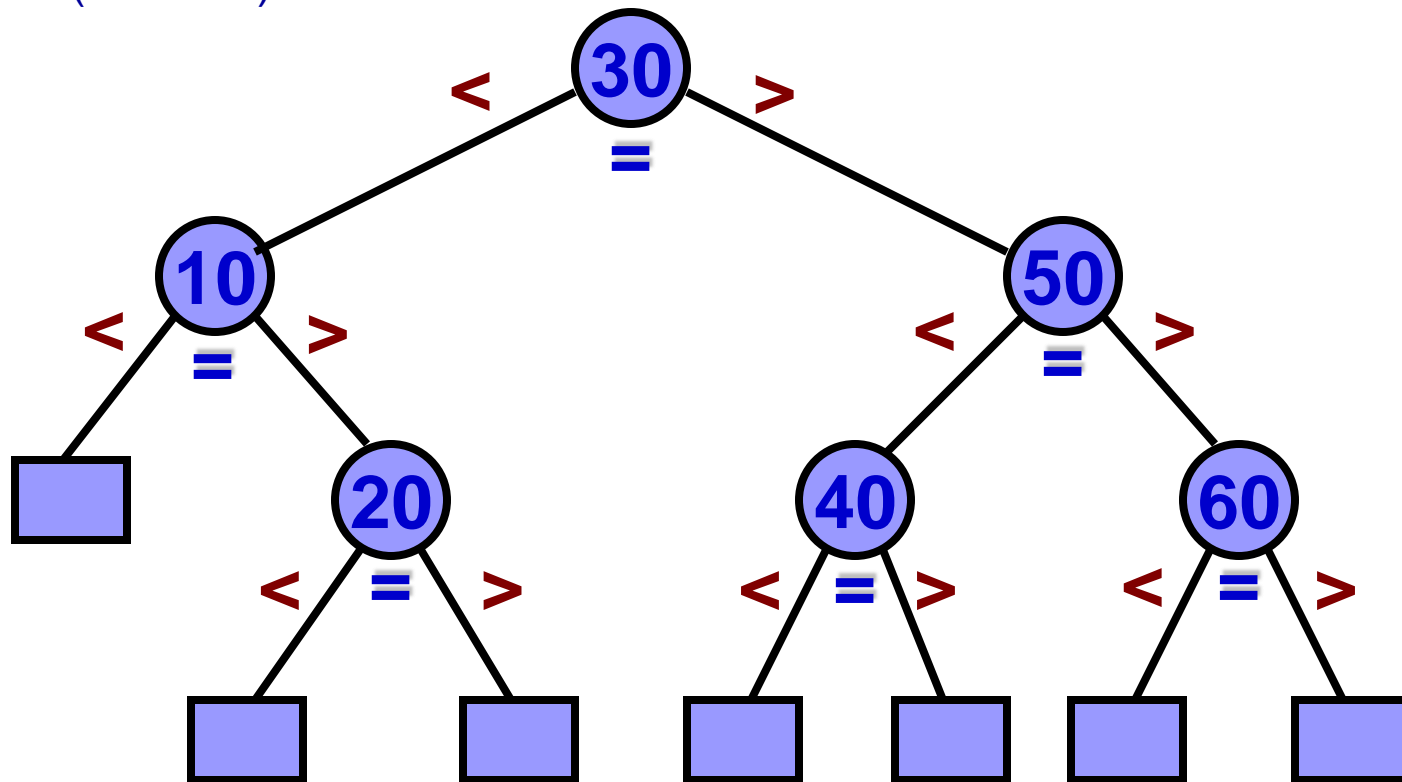
```
int binSearch(int[] a, int low, int high,int x) {  
    int mid=(low+high)/2;  
    if(low<=high) {  
        if(a[mid]<x) return binSearch(a,mid+1,high,x);  
        else if(a[mid]>x) return binSearch(a,low,mid-1,x);  
        else return mid;  
    }  
    return -1;  
} }
```



有序顺序表的折半搜索的判定树

(10, 20, 30, 40, 50, 60)

$$(2*1+3*6)/7$$



折半搜索性能分析

- 若设 $n = 2^h - 1$ ，则描述折半搜索的判定树是高度为 h 的满二叉树。

$$n = 2^h - 1, h = \log_2 \lfloor (n+1) \rfloor。$$

搜索成功：在表中查找任一记录的过程，即是折半查找判定树中从根结点到该记录结点的路径，和给定值的比较次数等于该记录结点在树中的层数。

搜索不成功：查找失败的过程就是走了一条从根结点到外部结点的路径，和给定值进行的关键码的比较次数等于该路径上内部结点的个数。

$$\begin{aligned}
 ASL_{succ} &= \sum_{i=0}^{n-1} p_i \cdot C_i = \frac{1}{n} \sum_{i=0}^{n-1} C_i = \frac{1}{n} (1 * 1 + 2 * 2^1 + \\
 &\quad + 3 * 2^2 + \dots + (h-1) \times 2^{h-2} + h \times 2^{h-1})
 \end{aligned}$$

可以用归纳法证明

$$\begin{aligned}
 &1 \times 1 + 2 \times 2^1 + 3 \times 2^2 + \dots + (h-1) \times 2^{h-2} + h \times 2^{h-1} = \\
 &= (h-1) \times 2^h + 1
 \end{aligned}$$

$$\begin{aligned}
 ASL_{succ} &= \frac{1}{n} ((h-1) \times 2^h + 1) = \frac{1}{n} ((n+1) \log_2(n+1) - n) \\
 &= \frac{n+1}{n} \log_2(n+1) - 1 \approx \log_2(n+1) - 1
 \end{aligned}$$



课堂练习（多项选择）：

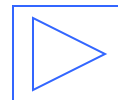
使用折半查找算法时，要求被查文件：

A．采用链式存贮结构

B．记录的长度 ≤ 128

☒ C．采用顺序存贮结构

☒ D．记录按关键字有序



动态查找表

特点：表结构在查找过程中动态生成。

要求：对于给定值key, 若表中存在其关键字等于key的记录, 则查找成功返回;

否则插入或删除关键字等于key的记录。

典型的动态表——二叉搜索树(二叉排序树)

一、二叉排序树的定义

二、二叉排序树的插入与删除

三、二叉排序树的查找分析

四、平衡二叉树

五、B-树



7.2 二叉搜索树 (Binary Search Tree)

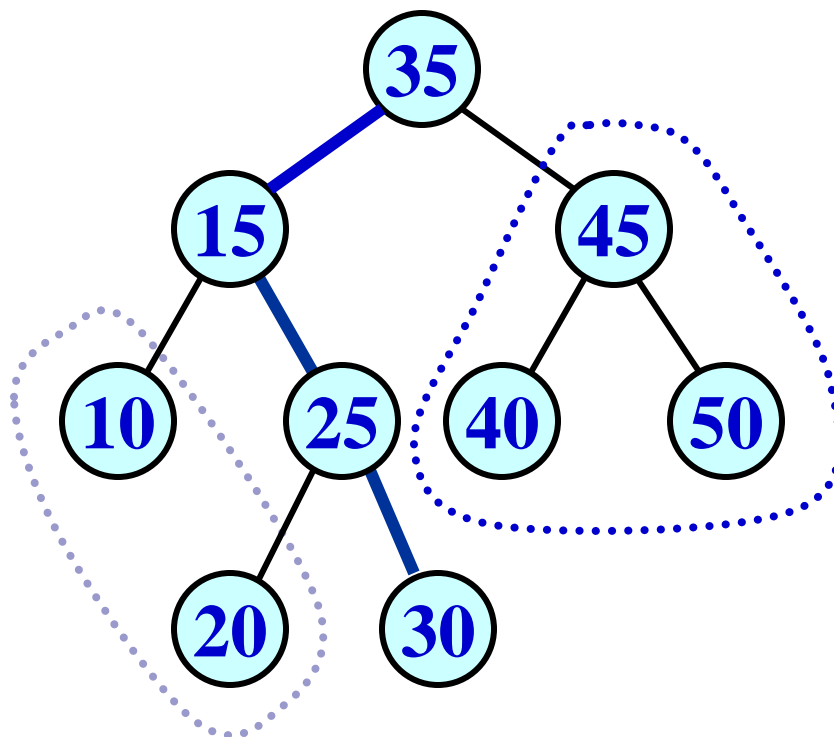
■ 定义

二叉搜索树或者是一棵空树，或者是具有下列性质的二叉树：

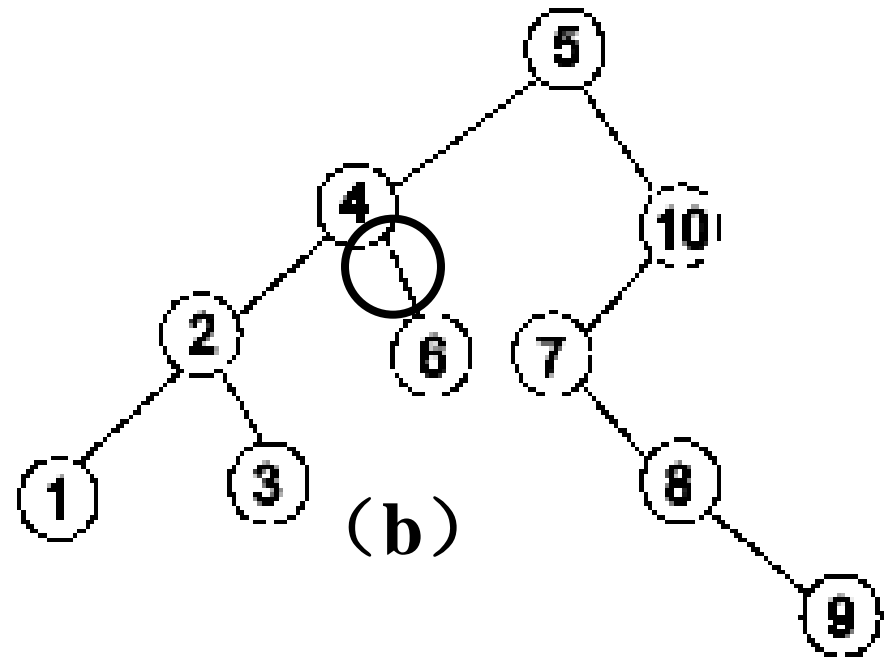
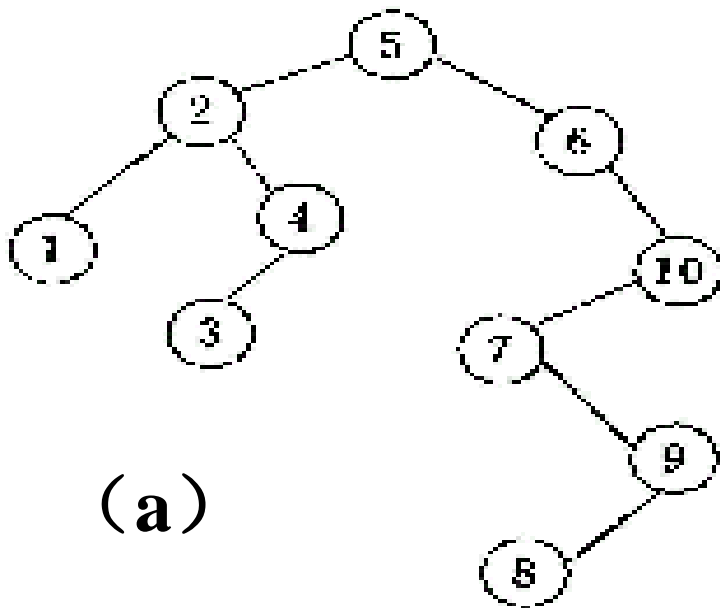
- ✓ 所有结点的键码互不相同。
- ✓ 左子树（如果非空）上所有结点的键码都小于根结点的键码。
- ✓ 右子树（如果非空）上所有结点的键码都大于根结点的键码。
- ✓ 左子树和右子树也是二叉搜索树。

二叉搜索树例

- 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；



练：下列2种图形中，哪个不是二叉搜索树？



- 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，所以也称二叉搜索树为二叉排序树。

- 二叉搜索树的类定义

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。

```
#include <iostream.h>
#include <stdlib.h>
template <class E, class K>
struct BSTNode {                                //二叉树结点类
    E data;                                       //数据域
    BSTNode<E, K> *left, *right; //左子女和右子女
```



```
BSTNode() { left = NULL; right = NULL; }
```

```
BSTNode (const E d, BSTNode<E, K> *L = NULL,  
        BSTNode<E, K> *R = NULL)
```

```
{ data = d; left = L; right = R;}
```

```
~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 K>
```

```
class BST {
```

```
private:
```

```
    BSTNode<E, K> *root;
```

```
    K RefValue;
```

```
public:
```

```
    BST() { root = NULL; }
```

```
    BST(K value);
```

```
    ~BST() {};
```

//二叉搜索树类定义

//根指针

//输入停止标志

//构造函数

//构造函数

//析构函数

```

bool Search (const K x) const           //搜索
    { return Search(x,root) != NULL; }

BST<E, K>& operator = (const BST<E, K>& R);
                                           //重载：赋值

void makeEmpty()                         //置空
    { makeEmpty (root); root = NULL;}

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)               //插入新元素
    { return Insert(e1, root);}

```



```
bool Remove (const K x) { return Remove(x, root);}
```

//删除含x的结点

private:

```
BSTNode<E, K> *           //递归：搜索
```

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

```
void makeEmpty (BSTNode<E, K> *& ptr);
```

//递归：置空

```
void PrintTree (BSTNode<E, K> *ptr) const;
```

//递归：打印

```
BSTNode<E, K> *           //递归：复制
```

```
    Copy (const BSTNode<E, K> *ptr);
```

BSTNode<E, K>* Min (BSTNode<E, K>* ptr);

//递归：求最小

BSTNode<E, K>* Max (BSTNode<E, K>* ptr);

//递归：求最大

bool Insert (const E& e1, BSTNode<E, K>* & ptr);

//递归：插入

bool Remove (const K x, BSTNode<E, K>* & ptr);

//递归：删除

};

二叉搜索树的搜索算法

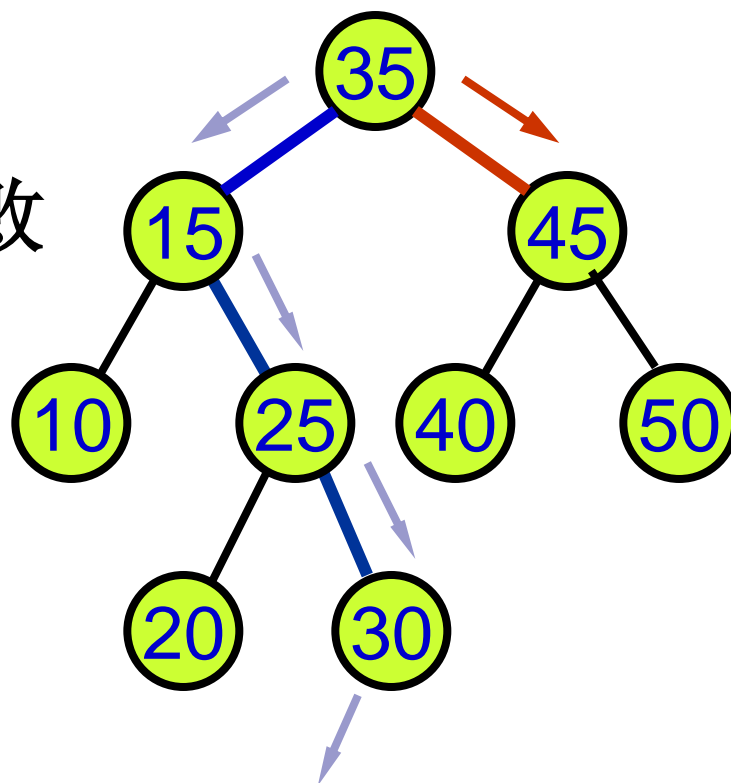
- 在二叉搜索树上进行搜索，是一个从根结点开始，递归进行比较判等的过程。
- 假设想要在二叉搜索树中搜索关键码为 x 的元素，搜索过程从根结点开始。
- 如果根指针为NULL，则搜索不成功；否则用给定值 x 与根结点的关键码进行比较：
 - ✓ 若给定值等于根结点关键码，则搜索成功
 - ✓ 若小于根结点的关键码，则搜索左子树；
 - ✓ 否则。递归搜索根结点的右子树。

搜索28

搜索失败

搜索45

搜索成功





```
template<class E, class K>
```

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

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

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

```
Search (const K x, BSTNode<E, K> *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, K>* temp = ptr;
```

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

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

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

```
    else temp = temp->right;
}
return NULL;
};
```

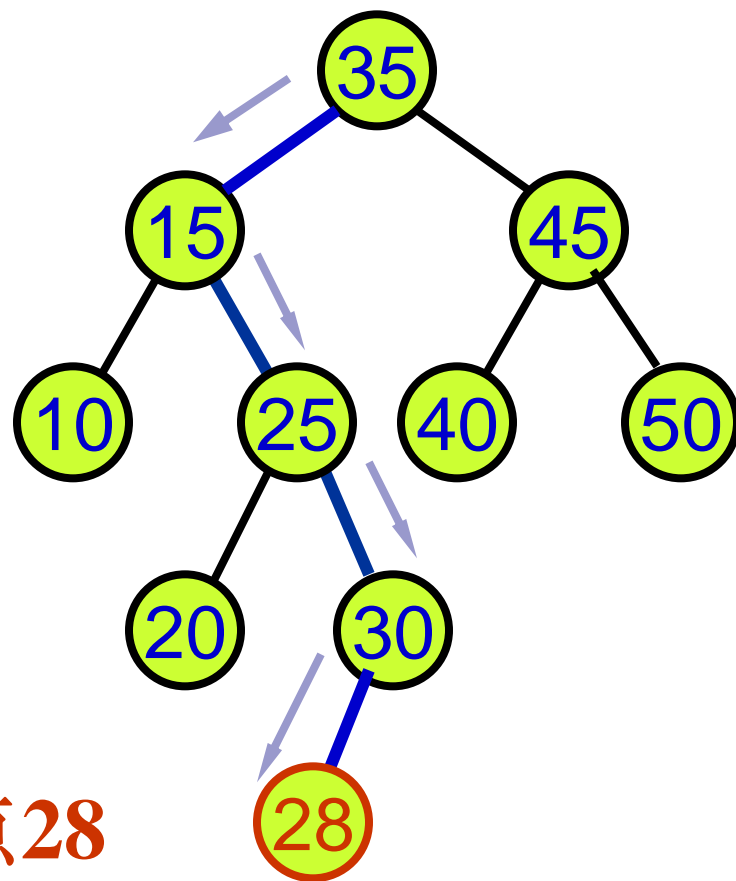
- 搜索过程是从根结点开始，**沿某条路径**自上而下逐层比较判等的过程。
- 搜索成功，搜索指针将停留在树上某个结点；搜索不成功，搜索指针将走到树上某个结点的空子树。
- 设树的高度为 h ，最多比较次数不超过 h 。

二叉搜索树的插入算法

- 为了向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。
 - 如果搜索成功，说明树中已经有这个元素，不再插入；
 - 如果搜索不成功，说明树中原来没有关键码等于给定值的结点，把新元素加到搜索操作停止的地方。

二叉搜索树的插入

- 每次结点的插入，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, class K>
```

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

```
{ //注意参数形式
```

```
    if (ptr == NULL) { //新结点作为叶结点插入
```

```
        ptr = new BstNode<E, K>(e1); //创建新结点
```

```
        if (ptr == NULL){ cerr << "Out of space" << endl; exit(1); }
```

```
        return true;
```


```
    }
```

```
    else if (e1 < ptr->data) return Insert (e1, ptr->left);
```

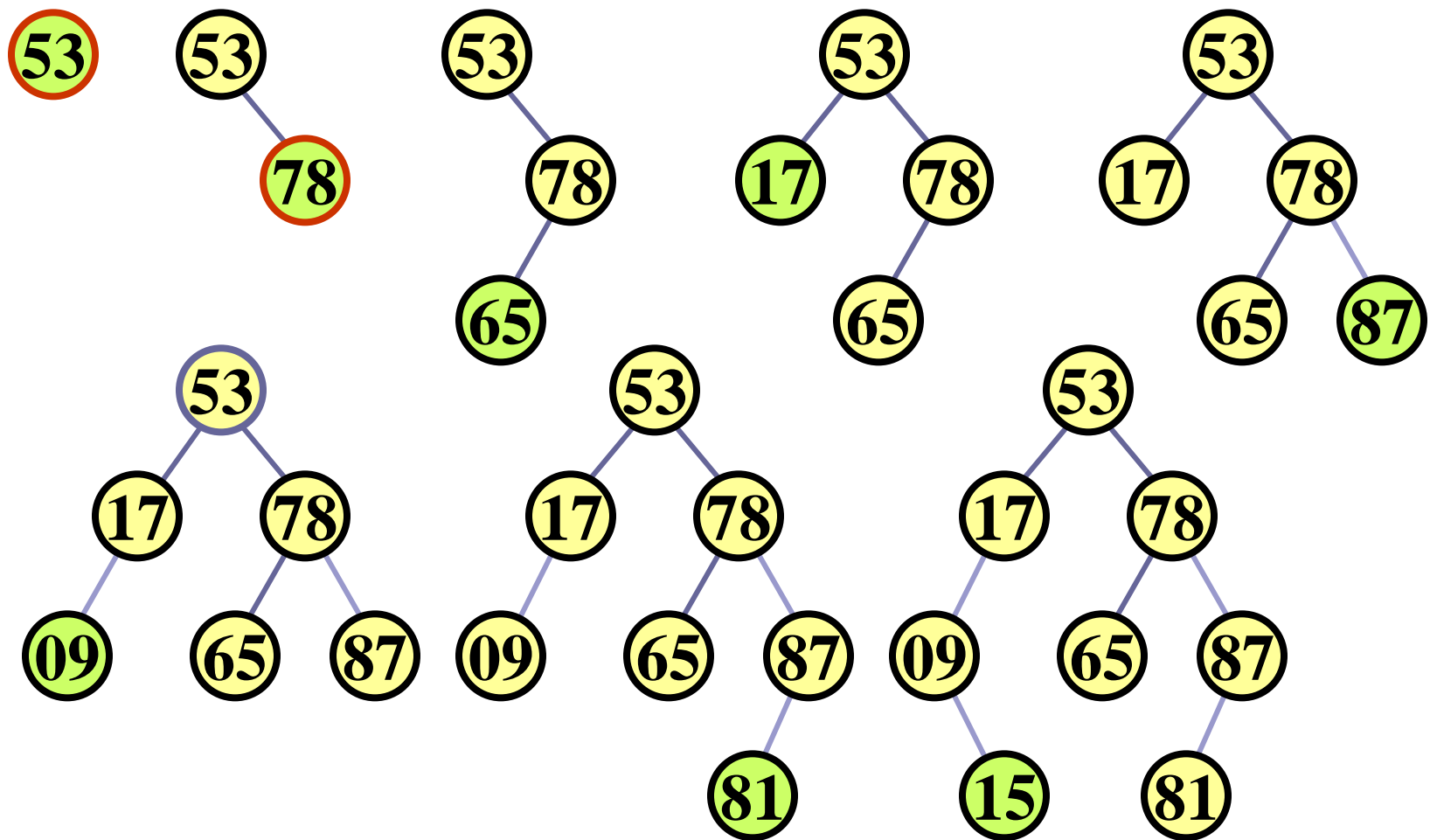
```
        else if (e1 > ptr->data) return Insert (e1, ptr->right);
```

```
            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, class K>
```

```
BST<E, K>::BST (K 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
```

```
    }
```

```
};
```

二叉搜索树插入的非递归方法（自学参考）

```
#include<iostream>
using namespace std;
typedef struct Binode {
    int data;
    Binode *l, *r;
}Binode,*Bitree;
```

```
void insert(Bitree & t, int i){
    Bitree p =t, parent;
    while (p != NULL){
        if (i > p->data){ parent=p;p = p->r;}
        else if (i < p->data) {parent=p;p = p->l;}
        else if (i == p->data) return;
    }
    p=new Binode; p->data = i; p->l = NULL; p->r = NULL;

    if(t==NULL) t=p;
    else
        if(parent->data<i) parent->r= p;
        else parent->l=p;
}
```

二叉搜索树插入的非递归方法（自学参考）

```
int main(){
```

```
    Bitree t = NULL;
```

```
    int n, k, i;
```

```
    cin >> n>> k;
```

```
    while (n--){
```

```
        cin >> i;
```

```
        insert(t, i);
```

```
        cout<<i<<" is inserted";
```

```
    }
```


二叉搜索树的删除

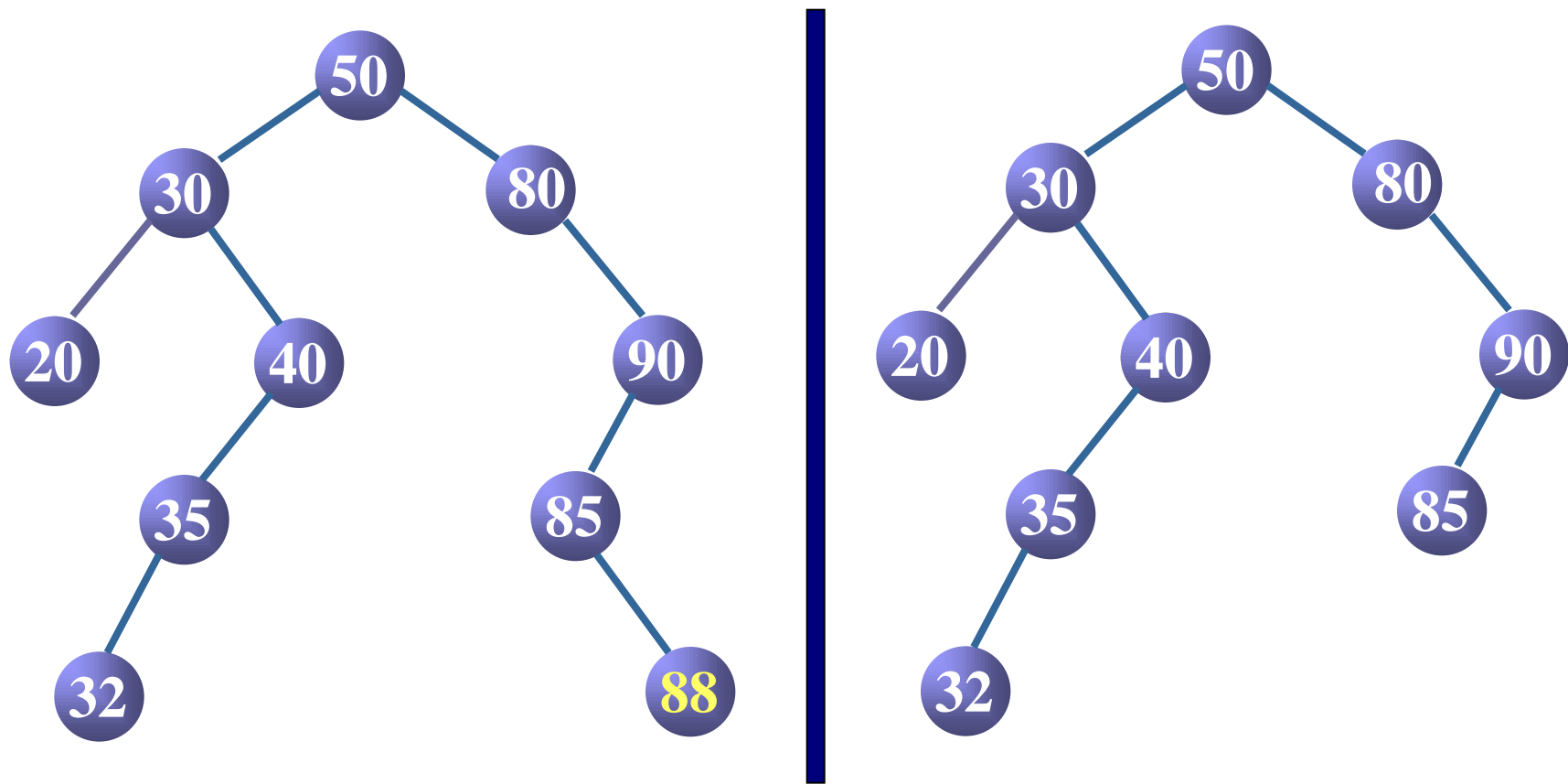
在二叉排序树上删除某个结点之后，仍然保持二叉排序树的特性。

分三种情况讨论：

- 被删除的结点是叶子；
- 被删除的结点只有左子树或者只有右子树；
- 被删除的结点既有左子树，也有右子树。

二叉搜索树的删除

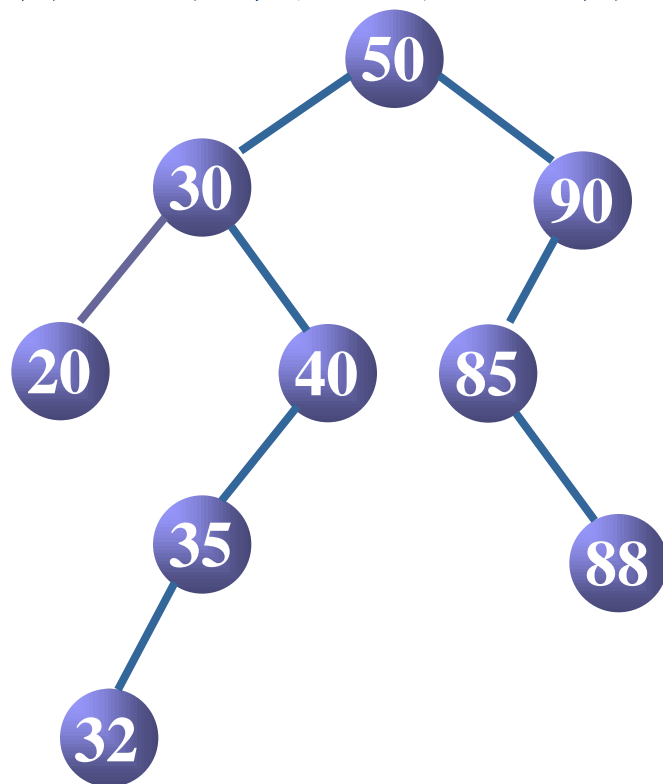
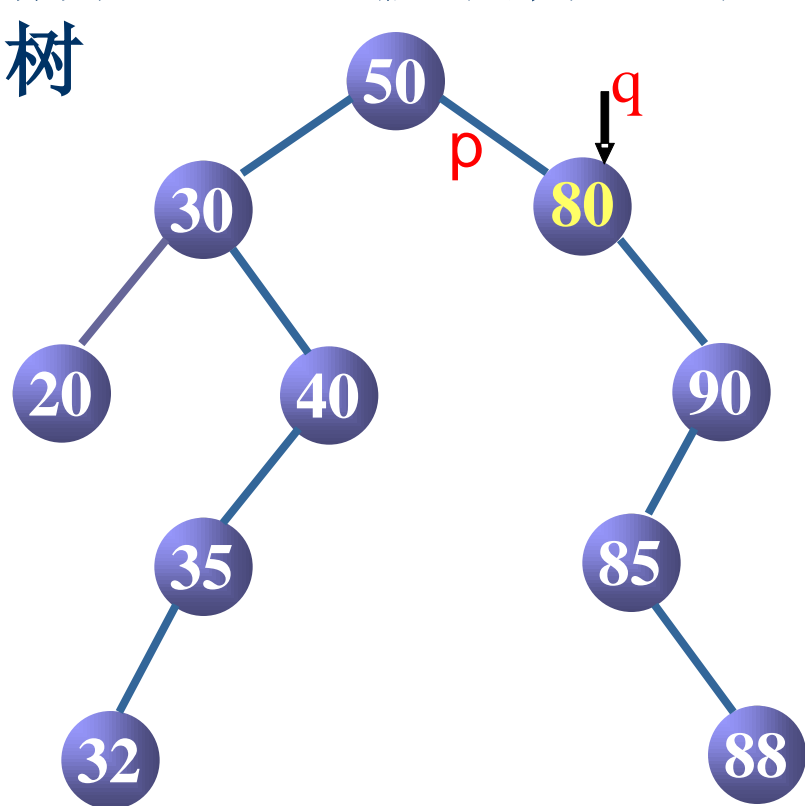
情况1——被删除的结点是叶子结点



操作：将双亲结点中相应指针域的值改为空。

二叉搜索树的删除

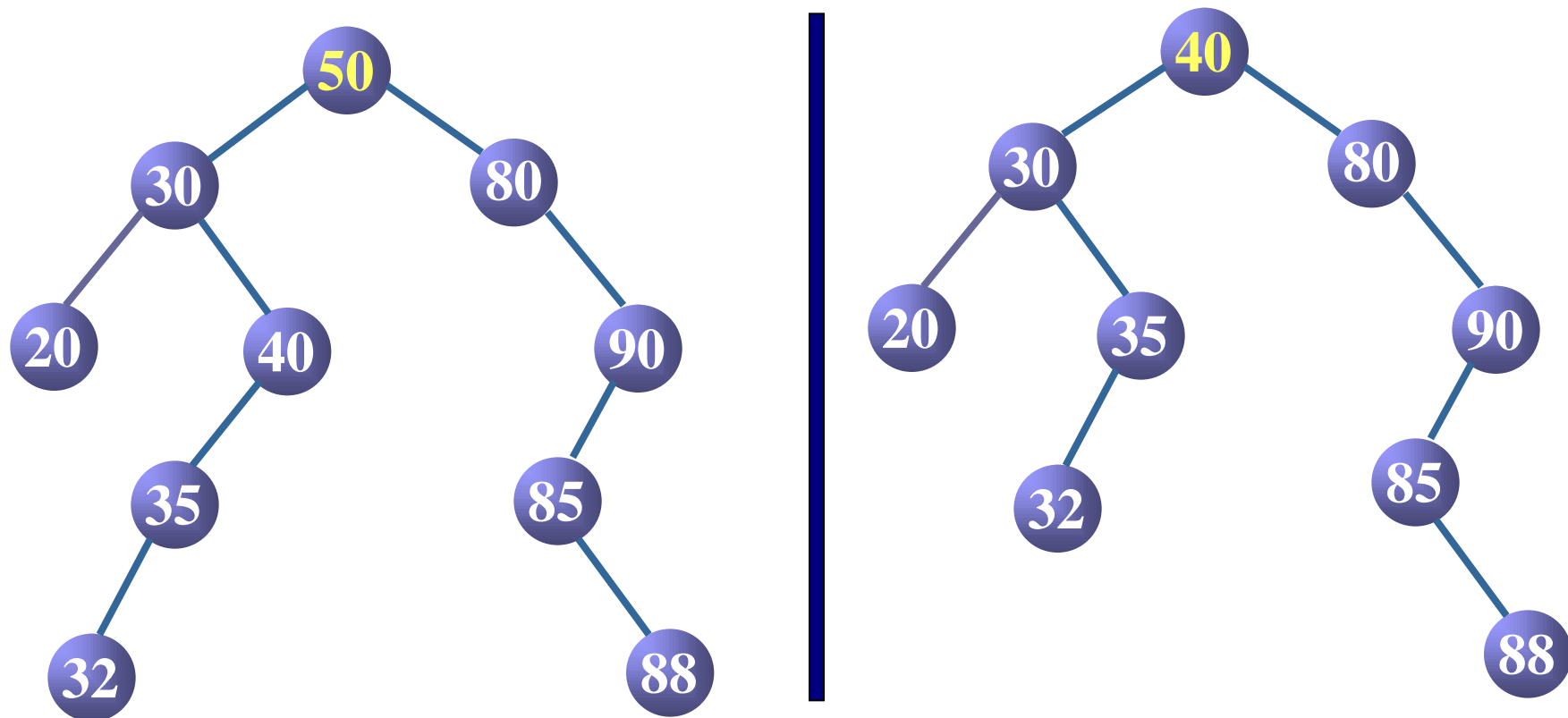
情况2——被删除的结点只有左子树或者只有右子树



操作：将双亲结点的相应指针域的值指向被删除结点的左子树（或右子树）。

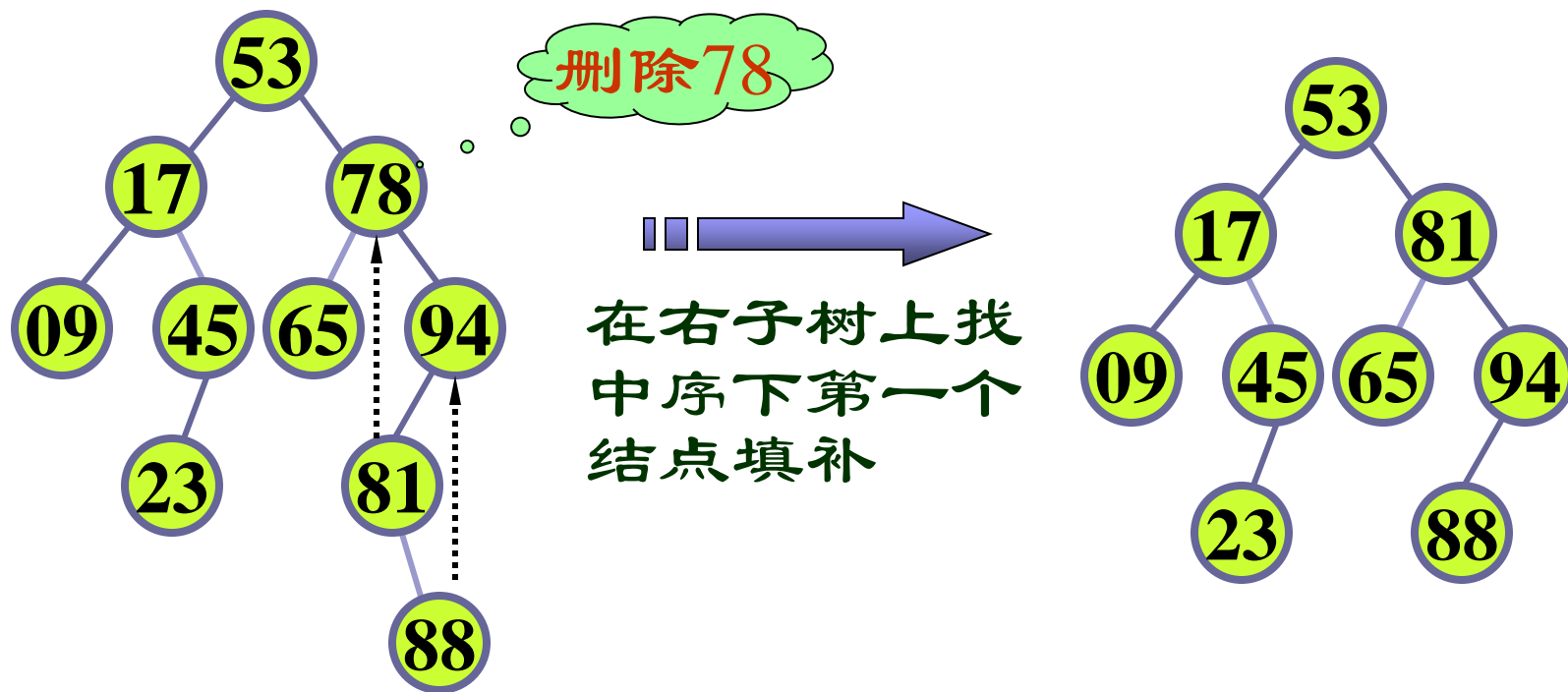
`q=p p=p->lchild（或p=p->rchild）； free(q)`

情况3: 被删结点左、右子树都不为空, 以其左子树中的最大值结点 (或右子树中的最小值结点) 替代之, 再来处理这个结点的删除问题。



二叉搜索树的删除

情况3——被删除的结点既有左子树也有右子树



二叉搜索树的删除算法

```
template <class E, class K>
```

```
bool BST<E, K>::Remove (const K x,
```

```
    BstNode<E, K> *& ptr) {
```

```
//在以 ptr 为根的二叉搜索树中删除含 x 的结点
```

```
    BstNode<E, K> *temp;
```

```
    if (ptr != NULL) {
```


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

```
                //在左子树中执行删除
```

```
        else if (x > ptr->data) return 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
    return 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。

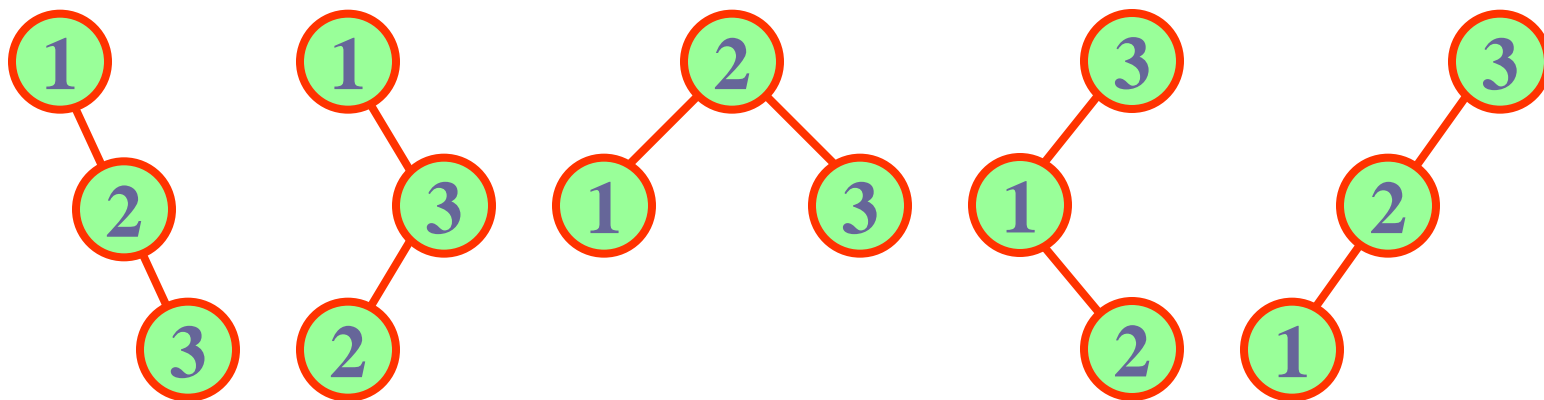
二叉搜索树性能分析

- 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

$$\frac{1}{n+1} C_{2n}^n \quad (\text{棵})$$

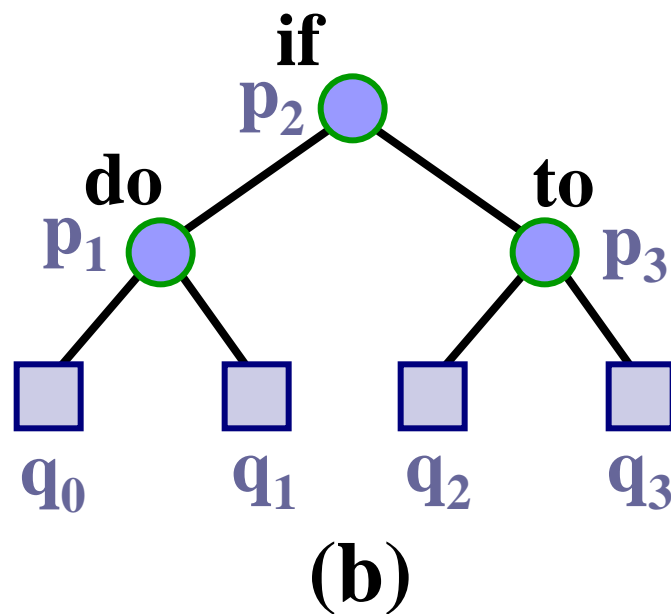
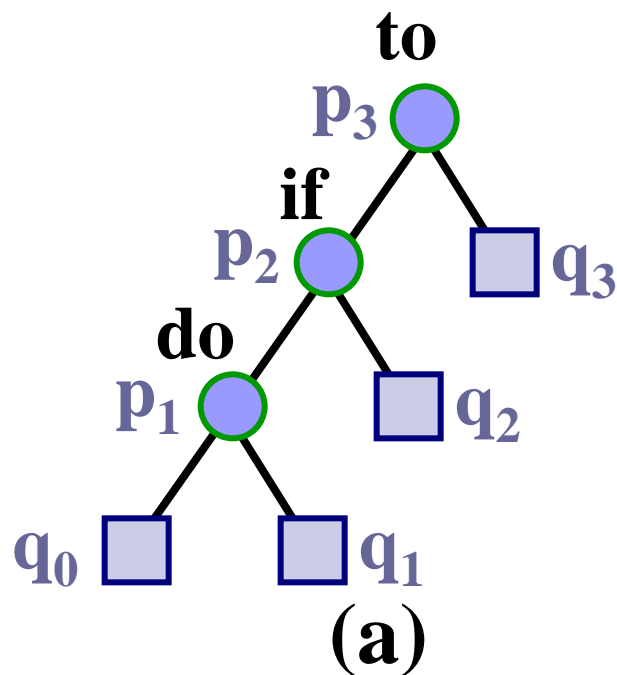
$\{2, 1, 3\}$

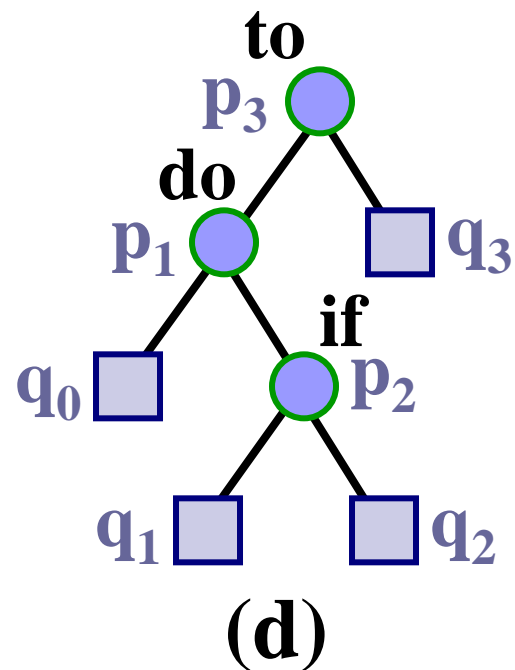
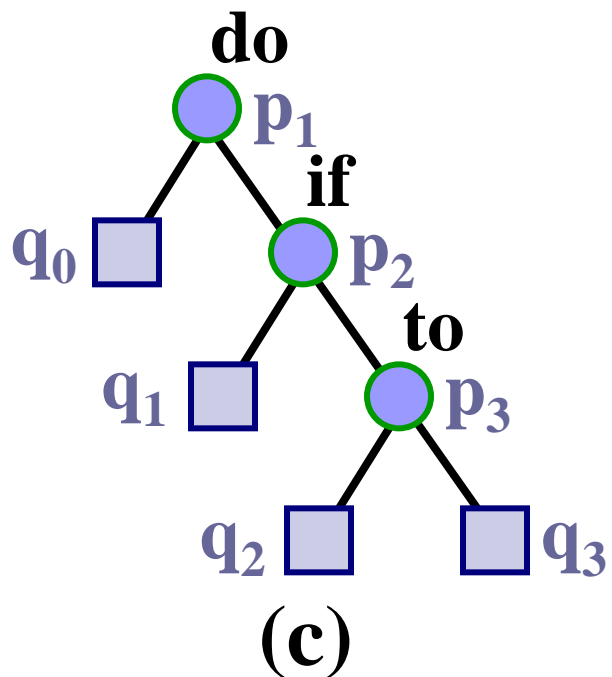
$\{1, 2, 3\}$ $\{1, 3, 2\}$ $\{2, 3, 1\}$ $\{3, 1, 2\}$ $\{3, 2, 1\}$



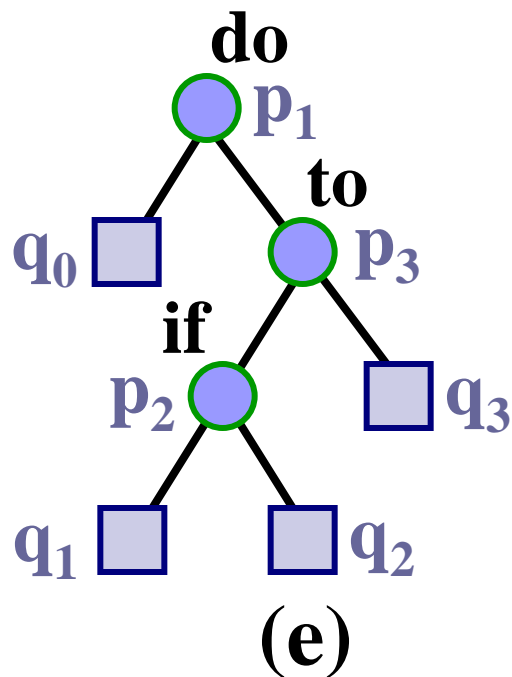
- 同样 3 个数据{ 1, 2, 3 }, 输入顺序不同, 建立起来的二叉搜索树的形态也不同。这直接影响到二叉搜索树的搜索性能。
- 如果输入序列选得不好, 会建立起一棵单支树, 使得二叉搜索树的高度达到最大。
- 在二叉搜索树中加入外结点, 形成判定树。外结点表示失败结点, 内结点表示搜索树中已有的数据。
- 这样的判定树即为扩充的二叉搜索树。

- 举例说明。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 ，在各搜索不成功间隔内搜索概率分别为 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。
- 一棵判定树上的搜索成功的平均搜索长度 ASL_{succ} 可以定义为该树所有内部结点上的搜索概率 $p[i]$ 与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:

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

- 设各关键码的搜索概率相等: $p[i] = 1/n$

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

- 搜索不成功的平均搜索长度 ASL_{unsucc} 为树中所有外部结点上搜索概率 $q[j]$ 与到达外部结点所需关键码比较次数 $c'[j](= l'[j])$ 乘积之和:

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

- 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): $ASL_{succ} = 1/3*1+1/3*3+1/3*2 = 6/3,$

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

- 图(b)的情形所得的平均搜索长度最小。
- 一般把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。同时，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 至少等于序列

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

的前 n 项的和。

- 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：

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

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

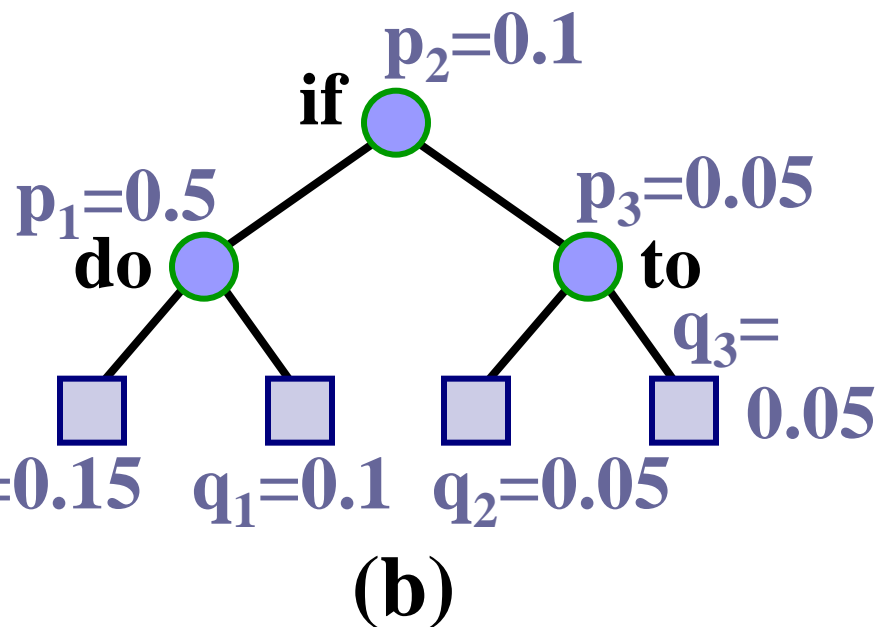
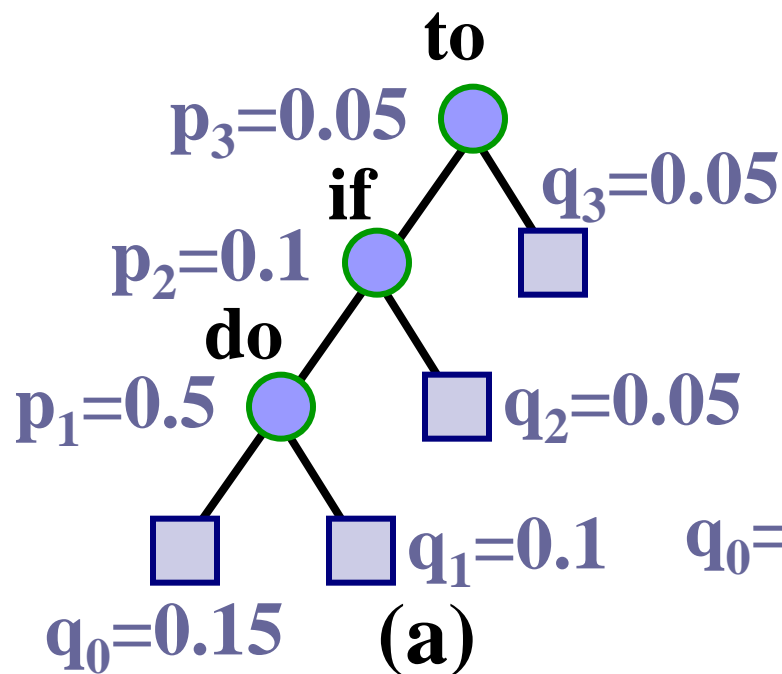
(2) 不相等搜索概率的情形

- 设二叉搜索树中所有内、外部结点的搜索概率互不相等。

$$p[1] = 0.5, p[2] = 0.1, p[3] = 0.05$$

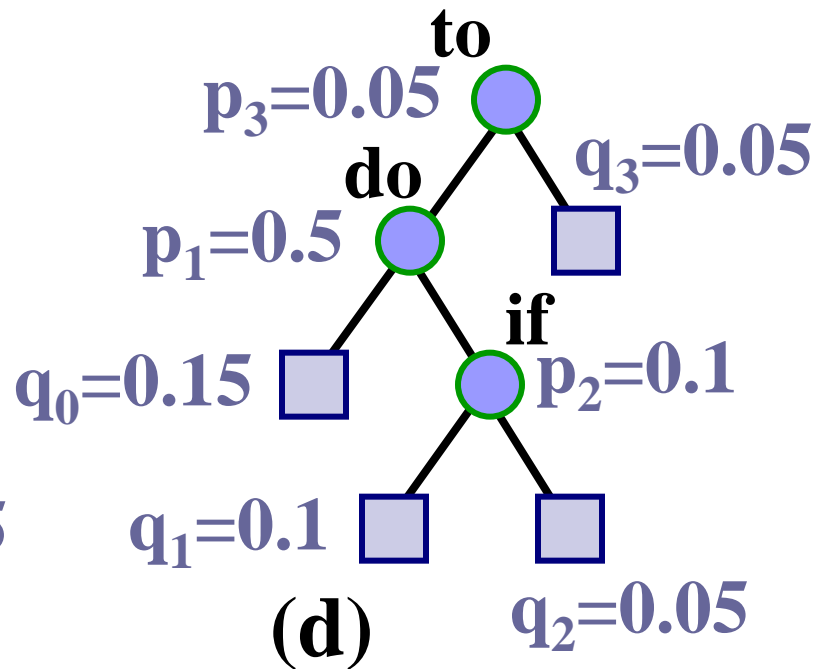
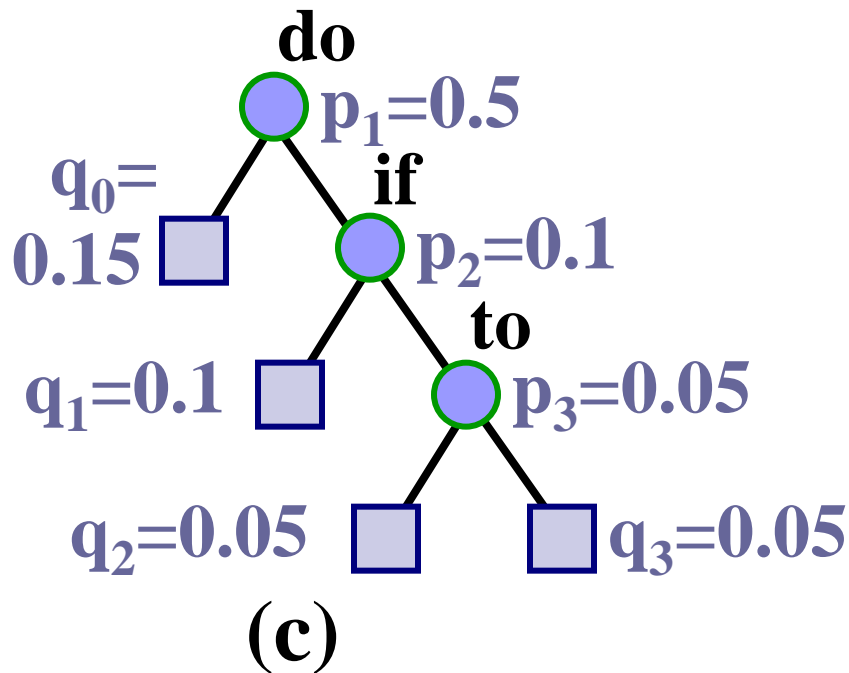
$$q[0] = 0.15, q[1] = 0.1, q[2] = 0.05, q[3] = 0.05$$

- 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): $ASL_{succ} = 0.5*2 + 0.1*1 + 0.05*2 = 1.2,$
 $ASL_{unsucc} = (0.15 + 0.1 + 0.05 + 0.05)*2 = 0.7。$



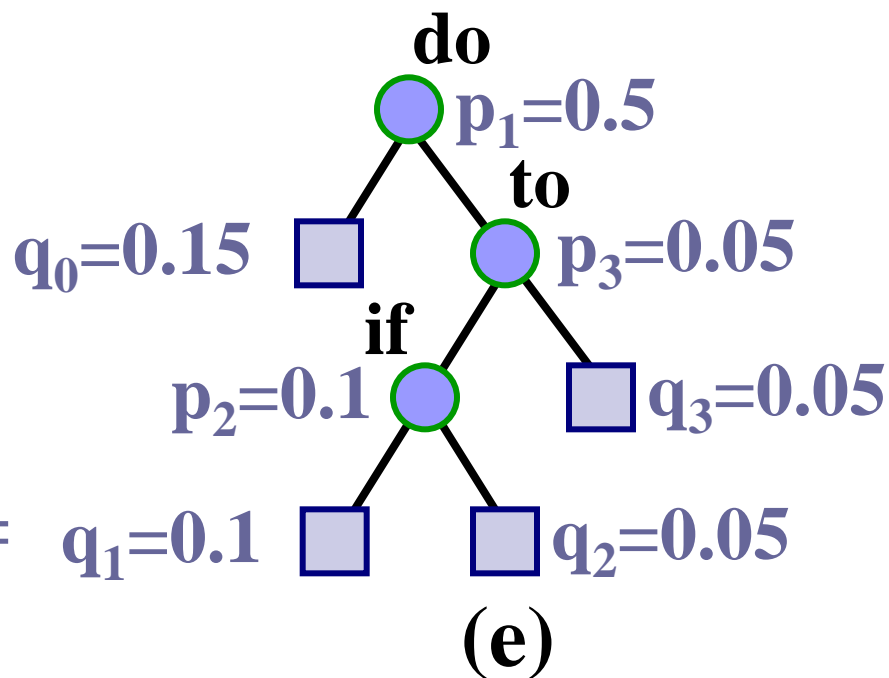
图(c): $ASL_{succ} = 0.5 * \mathbf{1} + 0.1 * \mathbf{2} + 0.05 * \mathbf{3} = 0.85,$
 $ASL_{unsucc} = 0.15 * \mathbf{1} + 0.1 * \mathbf{2} + 0.05 * \mathbf{3} + 0.05 * \mathbf{3}$
 $= 0.75.$

图(d) : $ASL_{succ} = 0.5 * \mathbf{2} + 0.1 * \mathbf{3} + 0.05 * \mathbf{1} = 1.35,$
 $ASL_{unsucc} = 0.15 * \mathbf{2} + 0.1 * \mathbf{3} + 0.05 * \mathbf{3} + 0.05 * \mathbf{1} = 0.8.$

图(e) :

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

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



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



本章总结

