法律声明

- 本课件包括演示文稿、示例、代码、题库、视频和声音等内容，小象学院和主讲老师拥有完全知识产权的权利；只限于善意学习者在本课程使用，不得在课程范围外向任何第三方散播。任何其他人或机构不得盗版、复制、仿造其中的创意及内容，我们保留一切通过法律手段追究违反者的权利。

- 课程详情请咨询
  - 微信公众号：小象
  - 新浪微博：ChinaHadoop

树

邹博

主要内容

- 二叉查找树
  - 增删改查/其他结构的基础：如AVL
- Huffman树
  - 前缀编码
- 二叉树的遍历
  - 前序中序后序三种遍历本身/通过前序中序求后序
  - 最大二叉搜索树
  - 二叉树的翻转
  - (隐式)树的搜索和应用
  - 所有括号匹配的字符串
  - K个不同字符的最长子串
- B树及其变种
  - 分裂结点/合并结点/R树
- 附：平衡二叉树

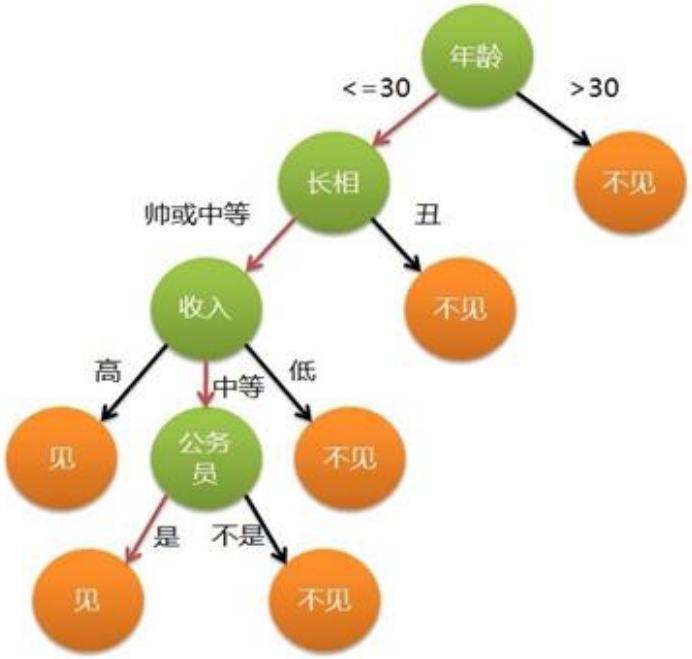
决策树的实例

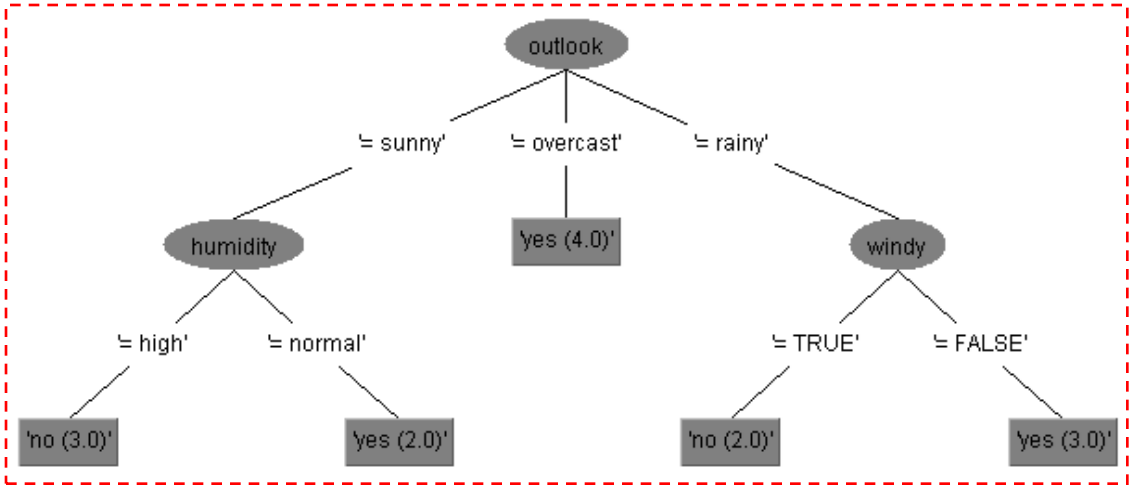
Viewer

Relation: weather.symbolic

No.	outlook Nominal	temperature Nominal	humidity Nominal	windy Nominal	play Nominal
1	sunny	hot	high	FALSE	no
2	sunny	hot	high	TRUE	no
3	overcast	hot	high	FALSE	yes
4	rainy	mild	high	FALSE	yes
5	rainy	cool	normal	FALSE	yes
6	rainy	cool	normal	TRUE	no
7	overcast	cool	normal	TRUE	yes
8	sunny	mild	high	FALSE	no
9	sunny	cool	normal	FALSE	yes
10	rainy	mild	normal	FALSE	yes
11	sunny	mild	normal	TRUE	yes
12	overcast	mild	high	TRUE	yes
13	overcast	hot	normal	FALSE	yes
14	rainy	mild	high	TRUE	no

Undo OK Cancel

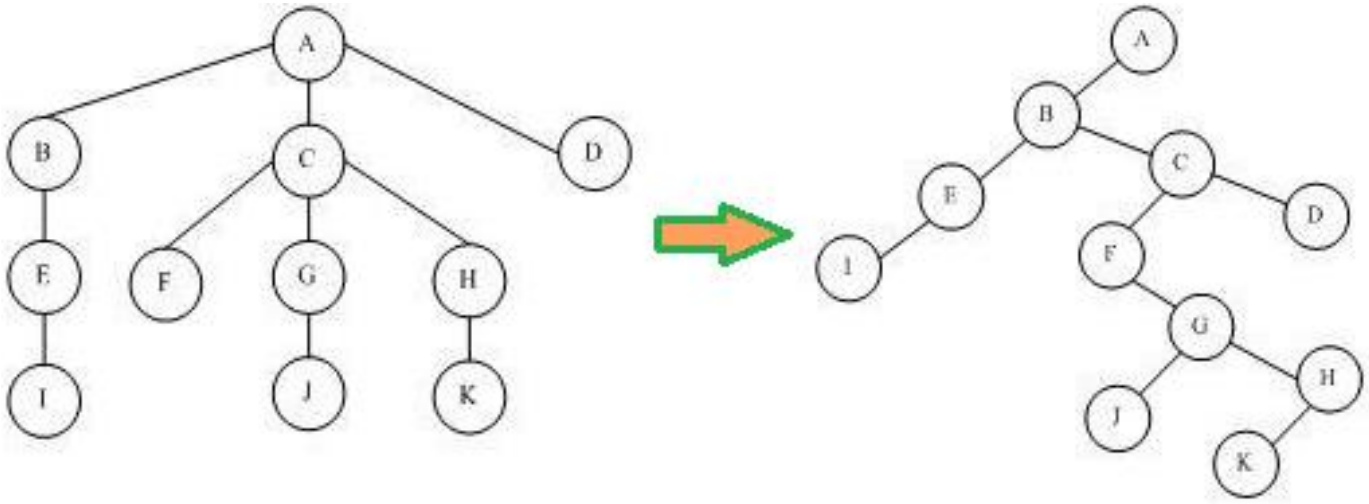




树和二叉树

- 一般地说，树的结点间是无序的，即：
- 一个结点有m个孩子$L_1L_2...L_m$，则$L_1L_2...L_m$可以互换位置，仍然认为是一颗树。
- 二叉树的两个孩子，一般称为左孩子、右孩子，不能互换位置。
  - 之所以这样定义，是因为有些算法，需要严格区分左右孩子，如前序-中序-后序遍历、堆排序等问题。
- 从这个意义上说，树和二叉树是两个概念，不能说二叉树是树的子集。
  - 注：这种区分非常弱而且乱，因为树还有“有序树”“无序树”的说法。

树转换成二叉树



树转换得到的二叉树右孩子

- 任意一颗树转换成二叉树，右孩子为空的数目为原树非叶结点+1。
  - 对于一颗树(非二叉树)，因为任何一个非叶结点必然有孩子，所以，它必然有最右孩子。从而，这个最右孩子转换到二叉树结点后，右指针必然为空。
  - 同时，根结点必然没有兄弟，所以，根结点转换成二叉树结点后，必然右孩子为空。
- 任意颗树转换成二叉树，右孩子为空的数目为原树非叶结点+1。
  - 如果是若干个树转二叉树，这若干个树最右边的那个树，是没有右孩子的。

关于树的思考题

- 现有一个包含m个节点的三叉树，即每个节点都有三个指向孩子节点的指针，请问：在这3m个指针中有_____个空指针。
- A:2m　B:2m-1　C:2m+1　D:3m

- 解析：
  - m个结点一共有3m个指针，m个结点的树一共有m-1的边，即m-1个指针是指向结点的，剩下的指针即为空指针：3m-(m-1)=2m+1

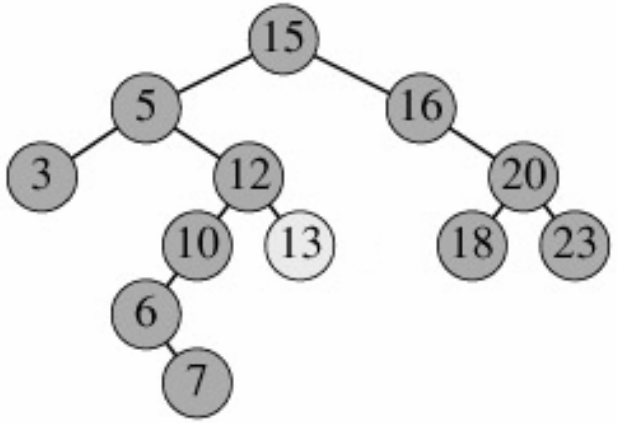
树基本操作

- 插入
- 删除
- 查找
- 前序遍历
- 中序遍历
- 后序遍历

```
typedef struct tagSTreeNode
{
    int value;
    tagSTreeNode* pLeft;
    tagSTreeNode* pRight;

    tagSTreeNode(int v) : value(v), pLeft(NULL), pRight(NULL) {}
}STreeNode;

typedef void (*VISIT)(int value);

class CBinaryTree
{
private:
    STreeNode* m_pRoot;

private:
    void Destroy(STreeNode* pRoot);
    bool _Insert(STreeNode*& pRoot, int value);              //递归
    bool _Insert2(int value);                                //非递归
    void _PreOrder(STreeNode* pRoot, VISIT Visit) const;     //递归
    void _PreOrder2(VISIT Visit) const;                      //非递归
    void _InOrder(STreeNode* pRoot, VISIT Visit) const;      //递归
    void _InOrder2(VISIT Visit) const;                       //非递归
    void _InOrder3(VISIT Visit) const;                       //非递归
    void _PostOrder(STreeNode* pRoot, VISIT Visit) const;    //递归
    void _PostOrder2(VISIT Visit) const;                     //非递归
    void DeleteChildless(STreeNode* pParent, STreeNode* pNode); //pNode无子
    void DeleteSingleSon(STreeNode* pParent, STreeNode* pNode); //pNode是pParent唯一子结点
public:
    CBinaryTree();
    ~CBinaryTree();
    bool Insert(int value);
    bool Delete(int value);
    STreeNode* Find(int value) const;
    void PreOrder(VISIT Visit) const;
    void InOrder(VISIT Visit) const;
    void PostOrder(VISIT Visit) const;
};
```

二叉查找树

- 二叉查找树(二叉搜索树)是满足以下条件的二叉树：
  - 左子树上的所有结点值均小于根结点值，
  - 右子树上的所有结点值均不小于根结点值，
  - 左右子树也满足上述两个条件



二叉查找树的查找

- 给定一颗二叉查找树，查找某结点p的过程如下：
  - 将当前结点cur赋值为根结点root；
  - 若p的值小于当前结点cur的值，查找cur的左子树；
  - 若p的值不小于当前结点cur的值，查找cur的右子树；
  - 递归上述过程，直到cur的值等于p的值或者cur为空；
    - 当然，若结点是结构体，注意定义“小于”“不小于”“等于”的具体函数。

Code

```
STreeNode* CBinaryTree::Find(int value) const
{
    if(!m_pRoot)
        return NULL;

    STreeNode* pNode = m_pRoot;
    while(pNode)
    {
        if(value < pNode->value)
            pNode = pNode->pLeft;
        else if(value > pNode->value)
            pNode = pNode->pRight;
        else
            return pNode;
    }
    return NULL;
}
```

二叉查找树的插入

- 插入过程如下：
  - 若当前的二叉查找树为空，则插入的元素为根结点，
  - 若插入的元素值小于根结点值，则将元素插入到左子树中，
  - 若插入的元素值不小于根结点值，则将元素插入到右子树中，
  - 递归上述过程，直到找到插入点为叶子结点。

Code (recursion)

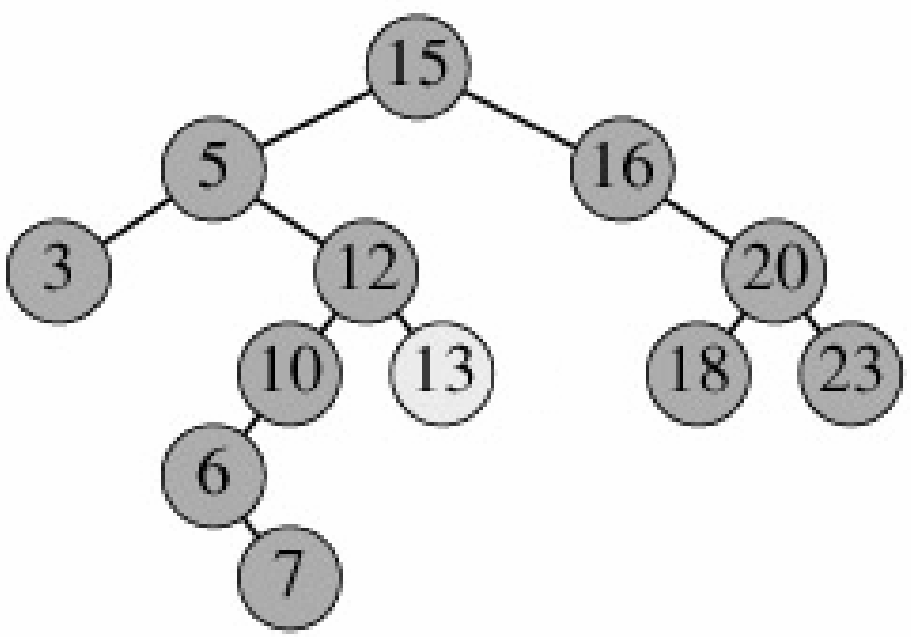
```
bool CBinaryTree::Insert(int value)
{
    return _Insert(m_pRoot, value);
}

bool CBinaryTree::_Insert(STreeNode*& pRoot, int value)
{
    if(!pRoot)
    {
        pRoot = new STreeNode(value);
        return true;
    }
    if(value < pRoot->value)
        return _Insert(pRoot->pLeft, value);
    if(value > pRoot->value)
        return _Insert(pRoot->pRight, value);
    return false;
}
```

Code

```
bool CBinaryTree::Insert(int value)
{
    return _Insert2(value);
}

bool CBinaryTree::_Insert2(int value)
{
    if(!m_pRoot)
    {
        m_pRoot = new STreeNode(value);
        return true;
    }
    STreeNode* pNode = m_pRoot;
    STreeNode* pCur = m_pRoot;
    while(pNode)
    {
        pCur = pNode;
        if(value < pNode->value)
            pNode = pNode->pLeft;
        else if(value > pNode->value)
            pNode = pNode->pRight;
        else
            return false;
    }
    if(value < pCur->value)
        pCur->pLeft = new STreeNode(value);
    else if(value > pCur->value)
        pCur->pRight = new STreeNode(value);
    return true;
}
```

二叉树的建立

- 依次插入：15,5,3,12,16,20,23,13,18,10,6,7

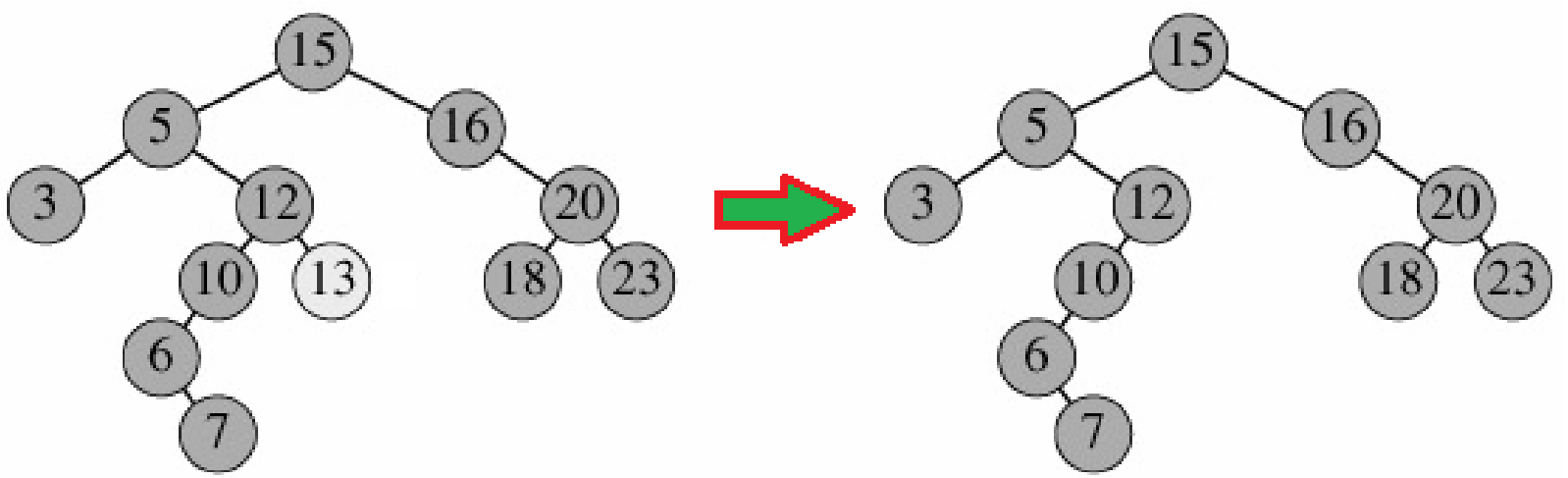


二叉查找树的删除

- 记待删除的结点为p，分三种情况进行处理：
  - p为叶子结点
  - p为单支结点
  - p的左子树和右子树均不空

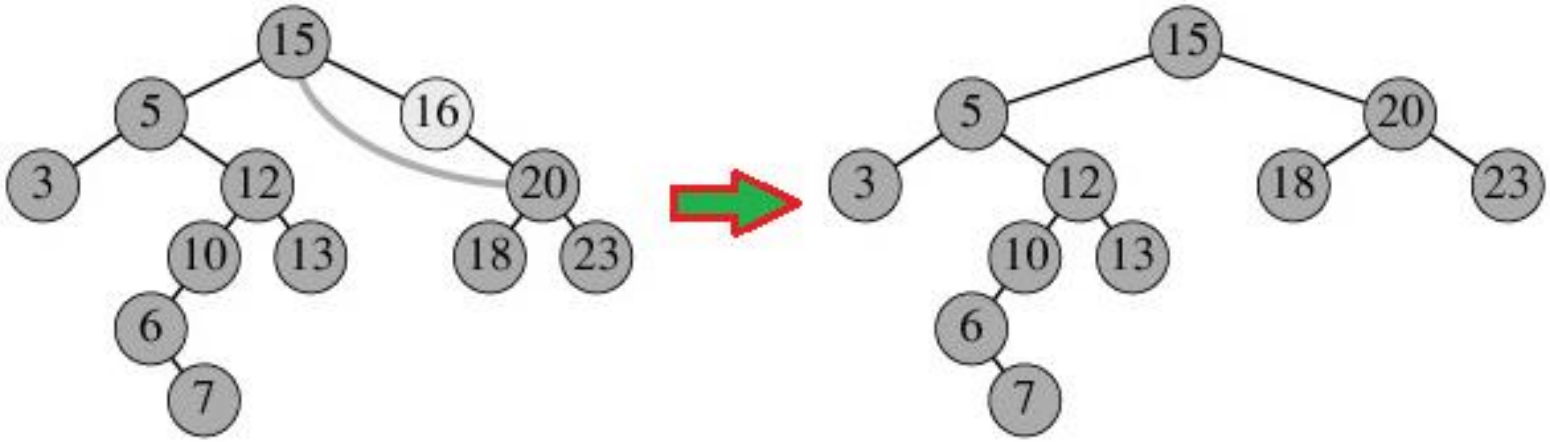
待删除点为叶子结点

- p为叶子结点，直接删除该结点，再修改p的父结点的指针

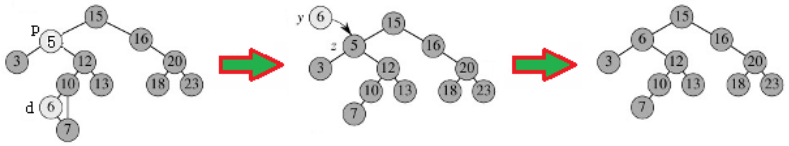


待删除点只有一个孩子

- 若p为单支结点(即只有左子树或右子树)，则将p的子树与p的父亲结点相连，删除p即可



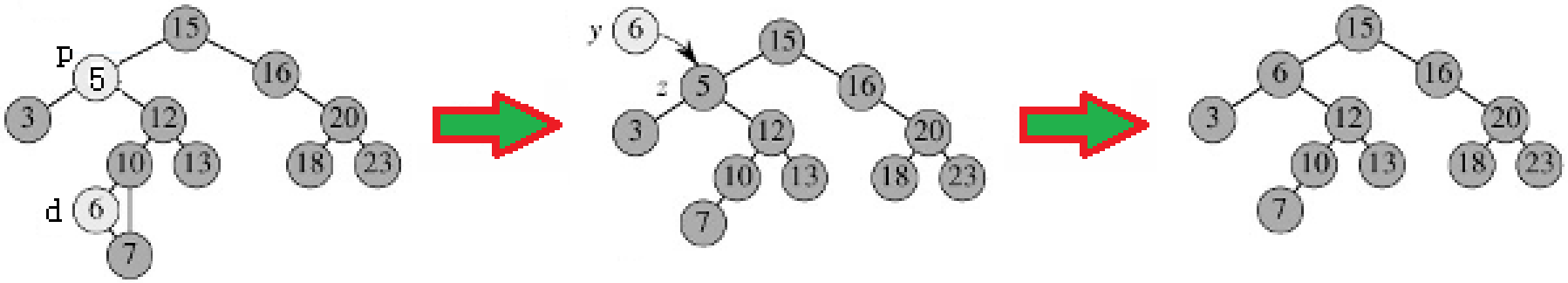
待删除点有两个孩子



- 若p的左子树和右子树均不空，则找到p的直接后继d(p的右孩子的最左子孙)，因为d一定没有左子树，所以使用删除单支结点的方法：删除d，并让d的父亲结点dp成为d的右子树的父亲结点；同时，用d的值代替p的值；
  - 对偶的，可以找到p的直接前驱x(p的左孩子的最右子孙)，x一定没有右子树，所以可以删除x，并让x的父亲结点成为x的左子树的父亲结点。

待删除点有两个孩子

- 任务：删除p
- 过程：两步走
  - 将p的直接后继的值拷贝到p处
  - 删除p的直接后继



Code

```
void CBinaryTree::DeleteChildless(STreeNode* pParent, STreeNode* pNode)
{
    if(pNode == m_pRoot)
        m_pRoot = NULL;
    else if(pParent->pLeft == pNode)
        pParent->pLeft = NULL;
    else
        pParent->pRight = NULL;
    delete pNode;
}

void CBinaryTree::DeleteSingleSon(STreeNode* pParent, STreeNode* pNode)
{
    STreeNode* pGrandSon = pNode->pLeft ? pNode->pLeft : pNode->pRight;
    if(pNode == m_pRoot)
        m_pRoot = pGrandSon;
    else if(pParent->pLeft == pNode)
        pParent->pLeft = pGrandSon;
    else
        pParent->pRight = pGrandSon;
    delete pNode;
}
```

Code

```
bool CBinaryTree::Delete(int value)
{
    if(!m_pRoot)
        return false;

    STreeNode* pNode = m_pRoot;
    STreeNode* pParent = NULL;
    while(pNode)
    {
        if(value < pNode->value)
        {
            pParent = pNode;
            pNode = pNode->pLeft;
        }
        else if(value > pNode->value)
        {
            pParent = pNode;
            pNode = pNode->pRight;
        }
        else    //找到待删除值
        {
            break;
        }
    }
    if(!pNode)  //没有找到
        return false;

    if(!pNode->pLeft && !pNode->pRight)
    {
        DeleteChildless(pParent, pNode);
    }
    else if(!pNode->pLeft || !pNode->pRight)
    {
        DeleteSingleSon(pParent, pNode);
    }
    else
    {
        STreeNode* pCur = pNode; //暂存待删除节点
        pParent = pNode;
        pNode = pNode->pLeft;
        while(pNode->pRight)
        {
            pParent = pNode;
            pNode = pNode->pRight;
        }
        pCur->value = pNode->value; //删除数据
        if(!pNode->pLeft)          //左孩子也空
            DeleteChildless(pParent, pNode);
        else
            DeleteSingleSon(pParent, pNode);
    }
    return true;
}
```

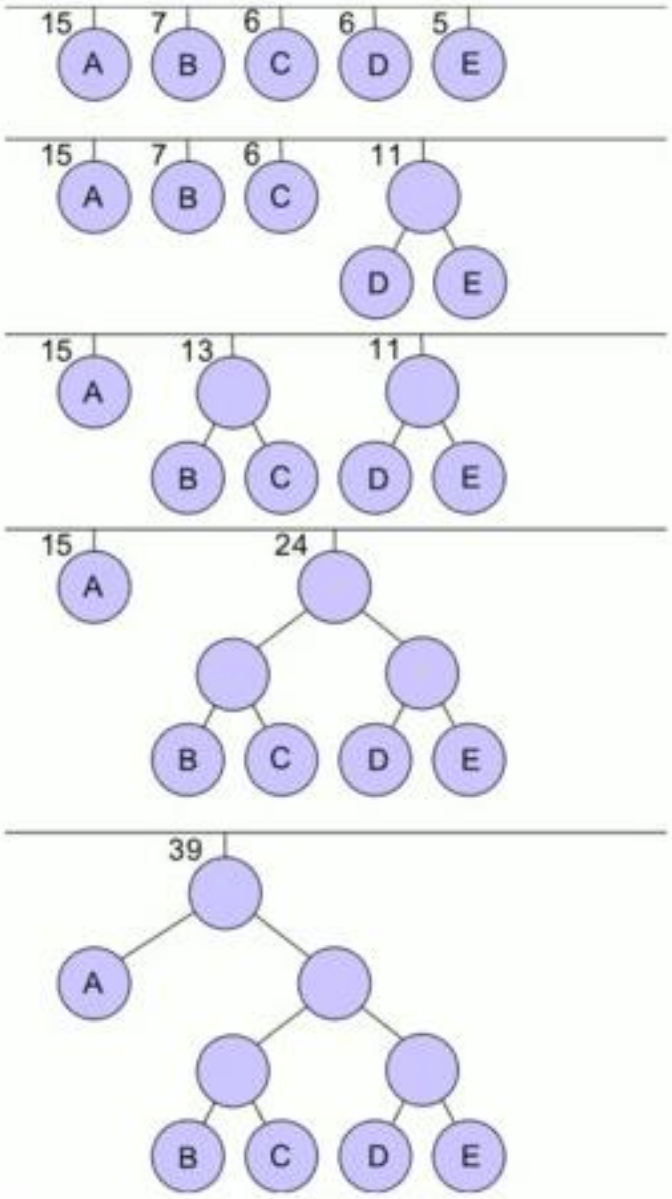
Code *split*

```
bool CBinaryTree::Delete(int value)
{
    if(!m_pRoot)
        return false;

    STreeNode* pNode = m_pRoot;
    STreeNode* pParent = NULL;
    while(pNode)
    {
        if(value < pNode->value)
        {
            pParent = pNode;
            pNode = pNode->pLeft;
        }
        else if(value > pNode->value)
        {
            pParent = pNode;
            pNode = pNode->pRight;
        }
        else    //找到待删除值
        {
            break;
        }
    }
    if(!pNode)  //没有找到
        return false;
    if(!pNode->pLeft && !pNode->pRight)
    {
        DeleteChildless(pParent, pNode);
    }
    else if(!pNode->pLeft || !pNode->pRight)
    {
        DeleteSingleSon(pParent, pNode);
    }
    else
    {
        STreeNode* pCur = pNode; //暂存待删除节点
        pParent = pNode;
        pNode = pNode->pLeft;
        while(pNode->pRight)
        {
            pParent = pNode;
            pNode = pNode->pRight;
        }
        pCur->value = pNode->value; //删除数据
        if(!pNode->pLeft)           //左孩子也空
            DeleteChildless(pParent, pNode);
        else
            DeleteSingleSon(pParent, pNode);
    }
    return true;
}
```

Huffman编码

- Huffman编码是一种无损压缩编码方案。
- 思想：根据源字符出现的(估算)概率对字符编码，概率高的字符使用较短的编码，概率低的使用较长的编码，从而使得编码后的字符串长度期望最小。
- Huffman编码是一种贪心算法：每次总选择两个最小概率的字符结点合并。
  - 称字符出现的次数为频数，则概率约等于频数除以字符总长；因此，概率可以用频数代替。

算法演示



Code

```
int _tmain(int argc, _TCHAR* argv[])
{
    const int N = 256;
    char str[] = "when I was young I'd listen to the radio\
                  waiting for my favorite songs\
                  when they played I'd sing along,\
                  it make me smile.\
                  those were such happy times and not so long ago\
                  how I wondered where they'd gone.\
                  but they're back again just like a long lost friend\
                  all the songs I love so well.\
                  every shalala every wo'wo\
                  still shines.\
                  every shing-a-ling-a-ling \
                  that they're starting\
                  to sing so fine";
    int pWeight[N] = {0};
    CalcFrequency(str, pWeight);
    pWeight['\t'] = 0;
    vector<int> pChar;
    CalcExistChar(pWeight, N, pChar);
    int N2 = (int)pChar.size();
    vector<vector<char> > code(N2);
    HuffmanCoding(pWeight, N2, code);
    Print(code, pChar);
    return 0;
}
```

Code

```
void CalcFrequency(const char* str, int* pWeight)
{
    while(*str)
    {
        pWeight[*str]++;
        str++;
    }
}

void CalcExistChar(int* pWeight, int N, vector<int>& pChar)
{
    int j = 0;
    for(int i = 0; i < N; i++)
    {
        if(pWeight[i] != 0)
        {
            pChar.push_back(i);
            if(j != i)
            {
                pWeight[j] = pWeight[i];
            }
            j++;
        }
    }
}
```

Main Code

```
void HuffmanCoding(int *pWeight, int N, vector<vector<char> >& code)
{
    if (N <= 0)
        return;
    int m = 2 * N - 1;  //N个结点的Huffman树需要2N-1个结点
    HuffmanNode* pHuffmanTree = new HuffmanNode[m];
    int s1, s2;

    int i;
    //建立叶子结点
    for (i = 0; i < N; i++)
        pHuffmanTree[i].nWeight = pWeight[i];

    //每次选择权值最小的两个结点，建树
    for(i = N; i < m; i++)
    {
        SelectNode(pHuffmanTree, i, s1, s2);
        pHuffmanTree[s1].nParent = pHuffmanTree[s2].nParent = i;
        pHuffmanTree[i].nLeft = s1;
        pHuffmanTree[i].nRight = s2;
        pHuffmanTree[i].nWeight = pHuffmanTree[s1].nWeight + pHuffmanTree[s2].nWeight;
    }

    //根据建好的Huffman树从叶子到根计算每个叶结点的编码
    int node, nParent;
    for(i = 0; i < N; i++)
    {
        vector<char>& cur = code[i];
        node = i;
        nParent = pHuffmanTree[node].nParent;
        while(nParent != 0)
        {
            if(pHuffmanTree[nParent].nLeft == node)
                cur.push_back('0');
            else
                cur.push_back('1');

            node = nParent;
            nParent = pHuffmanTree[node].nParent;
        }
        reverse(cur.begin(), cur.end());
    }
}
```

Aux Code

```
void SelectNode(const HuffmanNode* pHuffmanTree, int n, int& s1, int &s2)
{
    s1 = -1;    //无效值
    s2 = -1;
    int nMin1 = -1; //无效值
    int nMin2 = -1;
    for(int i = 0; i < n; i++)
    {
        if((pHuffmanTree[i].nParent == 0) && (pHuffmanTree[i].nWeight > 0))
        {
            if((s1 < 0) || (nMin1 > pHuffmanTree[i].nWeight))
            {
                s2 = s1;
                nMin2 = nMin1;
                s1 = i;
                nMin1 = pHuffmanTree[s1].nWeight;
            }
            else if((s2 < 0) || (nMin2 > pHuffmanTree[i].nWeight))
            {
                s2 = i;
                nMin2 = pHuffmanTree[s2].nWeight;
            }
        }
    }
}
```

Aux Code

```
void PrintCode(char c, vector<char>& code)
{
    cout << (int)c << "    " << c << ":    ";
    for(vector<char>::iterator it = code.begin(); it != code.end(); it++)
    {
        cout << *it;
    }
    cout << '\n';
}

void Print(vector<vector<char> >& code, vector<int>& pChar)
{
    int size = (int)code.size();
    for(int i = 0; i < size; i++)
    {
        PrintCode(pChar[i], code[i]);
    }
}
```

实验结果

```
when I was young I'd listen to the radio
waiting for my favorite songs
when they played I'd sing along,
it make me smile.
those were such happy times and not so long ago
how I wondered where they'd gone.
but they're back again just like a long lost friend
all the songs I love so well.
every shalala every wo'wo
still shines.
every shing-a-ling-a-ling
that they're starting
to sing so fine
```

32	:	110	106	j:	01010101
39	':	111000	107	k:	0101011
44	,:	01010100	108	l:	0010
45	-:	1110011	109	m:	001111
46	.:	1111000	110	n:	1001
73	I:	001110	111	o:	1010
97	a:	0110	112	p:	1110010
98	b:	11110110	114	r:	10111
99	c:	11110111	115	s:	0111
100	d:	00110	116	t:	1000
101	e:	000	117	u:	1111010
102	f:	1111001	118	v:	010100
103	g:	11101	119	w:	01011
104	h:	11111	121	y:	10110
105	i:	0100			

Huffman编码总结：前缀编码

- Huffman编码是不等长编码
  - 字符的编码长度不完全相同。
- 不等长编码如果需要译码，必须满足“前缀编码”的条件：任何一个字符的编码都不是另外一个字符编码的前缀。
- 字符串：ABBC
- 使用编码方案：
  - A:0　　B:1　　C:00
- 则，ABBC的 编码为：01100
- 01100的译码可以是ABBC，也可以是ABBAA。

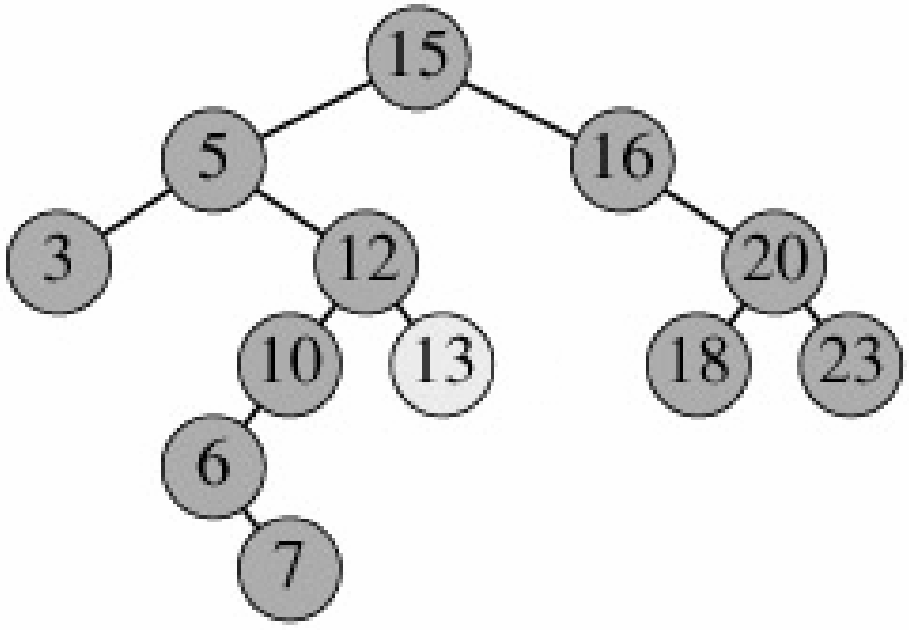
Huffman实现带来的思考

- Huffman编码是如何解决前缀编码问题的？
- 实际算法往往是由多个“小算法”堆砌而成的。
  - 空格压缩问题
  - 取数组最大/小的两个数
    - 进一步思考：有更快的算法么？(优先队列)
- 代码实现中并非直接使用指针形成的二叉树结点。而是事先开辟足够大的缓冲空间(2n+1)，每次从缓冲区获取一个结点，使用数组代替二叉树。
  - 在堆排序、双数组Trie树结构等问题中会再次遇到。
- 由于Huffman树的结点权值(频数)可能相等，因此，对某些文本，Huffman编码不唯一。
  - “左赋1，右赋0”或者“左赋0，右赋1”都可以。

二叉树的遍历

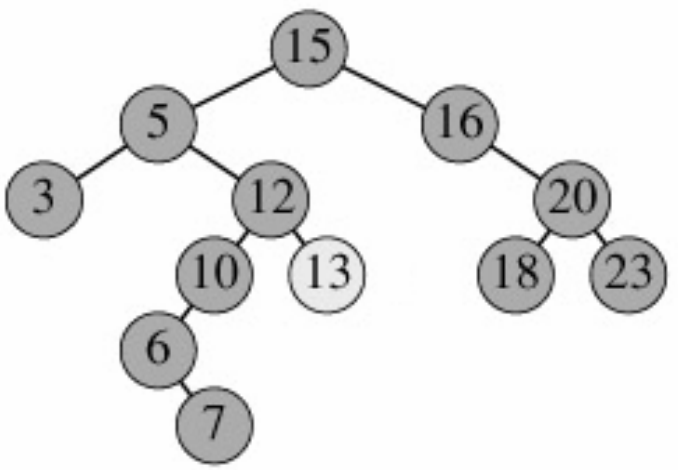
- 前序遍历：
  - 访问根结点
  - 前序遍历左子树
  - 前序遍历右子树
- 中序遍历：
  - 中序遍历左子树
  - 访问根结点
  - 中序遍历右子树
- 后序遍历：
  - 后序遍历左子树
  - 后序遍历右子树
  - 访问根结点

前序遍历



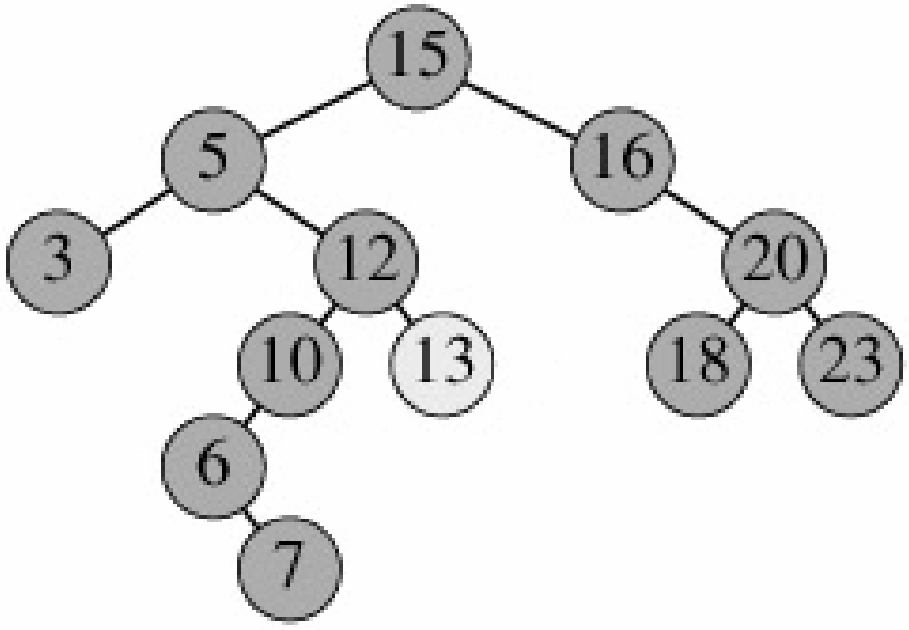
- 前序遍历：15,5,3,12,10,6,7,13,16,20,18,23

Code



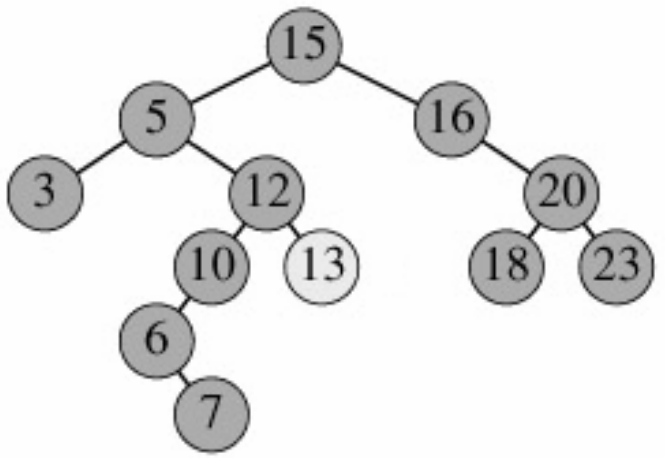

```
void CBinaryTree::PreOrder(VISIT Visit) const
{
    //_PreOrder(m_pRoot, Visit);
    _PreOrder2(Visit);
}

void CBinaryTree::_PreOrder(STreeNode* pRoot, VISIT Visit) const
{
    if(pRoot)
    {
        Visit(pRoot->value);
        _PreOrder(pRoot->pLeft, Visit);
        _PreOrder(pRoot->pRight, Visit);
    }
}

void CBinaryTree::_PreOrder2(VISIT Visit) const
{
    if(!m_pRoot)
        return;

    stack<STreeNode*> s;
    s.push(m_pRoot);
    STreeNode* pCur;
    while(!s.empty())
    {
        pCur = s.top();
        s.pop();
        Visit(pCur->value);
        if(pCur->pRight)
            s.push(pCur->pRight);
        if(pCur->pLeft)
            s.push(pCur->pLeft);
    }
}
```

中序遍历



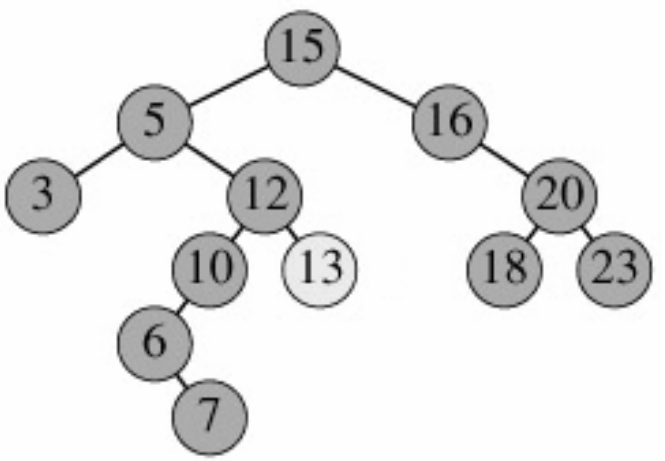
- 中序遍历：3,5,6,7,10,12,13,15,16,18,20,23
  - 二叉查找树的中序遍历，即为数据的升序过程

Code



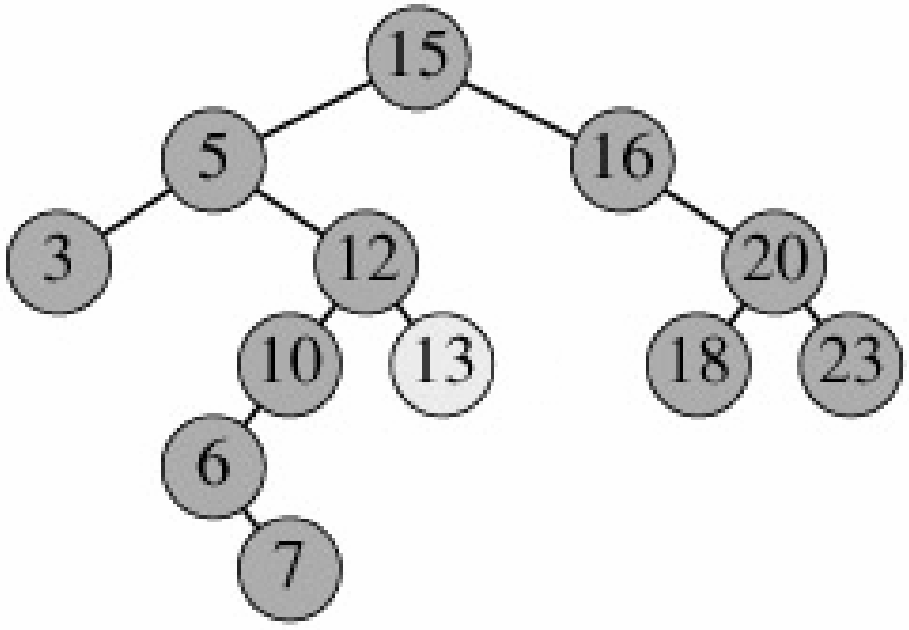

```
void CBinaryTree::InOrder(VISIT Visit) const
{
    _InOrder(m_pRoot, Visit);
    //_InOrder2(Visit);
}

void CBinaryTree::_InOrder(STreeNode* pRoot, VISIT Visit) const
{
    if(pRoot)
    {
        _InOrder(pRoot->pLeft, Visit);
        Visit(pRoot->value);
        _InOrder(pRoot->pRight, Visit);
    }
}

void CBinaryTree::_InOrder2(VISIT Visit) const
{
    stack<STreeNode*> s;
    STreeNode* pCur = m_pRoot;
    while(pCur || !s.empty())
    {
        while(pCur) //找最左孩子
        {
            s.push(pCur);
            pCur = pCur->pLeft;
        }
        if(!s.empty())
        {
            pCur = s.top(); //访问左孩子为空的结点
            s.pop();
            Visit(pCur->value);
            pCur = pCur->pRight;    //转向右孩子
        }
    }
}
```

Code




```
void CBinaryTree::_InOrder3(VISIT Visit) const
{
    if(!m_pRoot)
        return;

    stack<pair<STreeNode*, int> > s;
    s.push(make_pair(m_pRoot, 0));
    int times;
    STreeNode* pCur;
    while(!s.empty())
    {
        pCur = s.top().first;
        times = s.top().second;
        s.pop();
        if(times == 0)  //第一次压栈
        {
            if(pCur->pRight)
                s.push(make_pair(pCur->pRight, 0));
            s.push(make_pair(pCur, 1)); //第二次压栈
            if(pCur->pLeft)
                s.push(make_pair(pCur->pLeft, 0));
        }
        else
        {
            Visit(pCur->value);
        }
    }
}
```

后序遍历

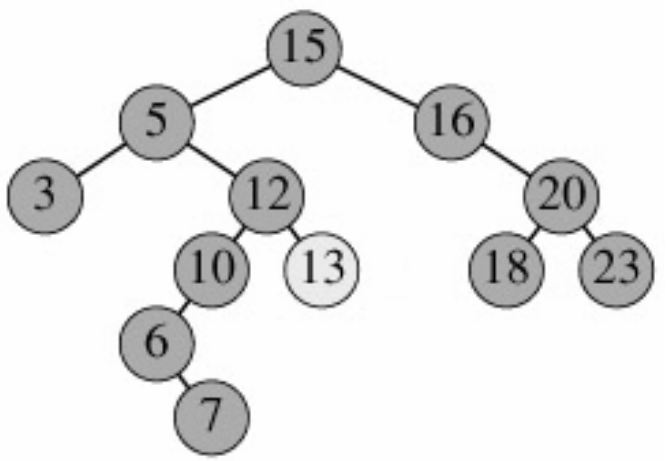


- 后序遍历：3,7,6,10,13,12,5,18,23,20,16,15

Code

```
void CBinaryTree::PostOrder(VISIT Visit) const
{
    _PostOrder(m_pRoot, Visit);
    //_PostOrder2(Visit);
}

void CBinaryTree::_PostOrder(STreeNode* pRoot, VISIT Visit) const
{
    if(pRoot)
    {
        _PostOrder(pRoot->pLeft, Visit);
        _PostOrder(pRoot->pRight, Visit);
        Visit(pRoot->value);
    }
}
```

Code




```
void CBinaryTree::_PostOrder2(VISIT Visit) const
{
    if(!m_pRoot)
        return;

    stack<pair<STreeNode*, int> > s;
    s.push(make_pair(m_pRoot, 0));
    int times;
    STreeNode* pCur;
    while(!s.empty())
    {
        pCur = s.top().first;
        times = s.top().second;
        s.pop();
        if(times == 0)  //第一次压栈
        {
            s.push(make_pair(pCur, 1)); //第二次压栈
            if(pCur->pRight)
                s.push(make_pair(pCur->pRight, 0));
            if(pCur->pLeft)
                s.push(make_pair(pCur->pLeft, 0));
        }
        else
        {
            Visit(pCur->value);
        }
    }
}
```

根据前序中序，计算后序

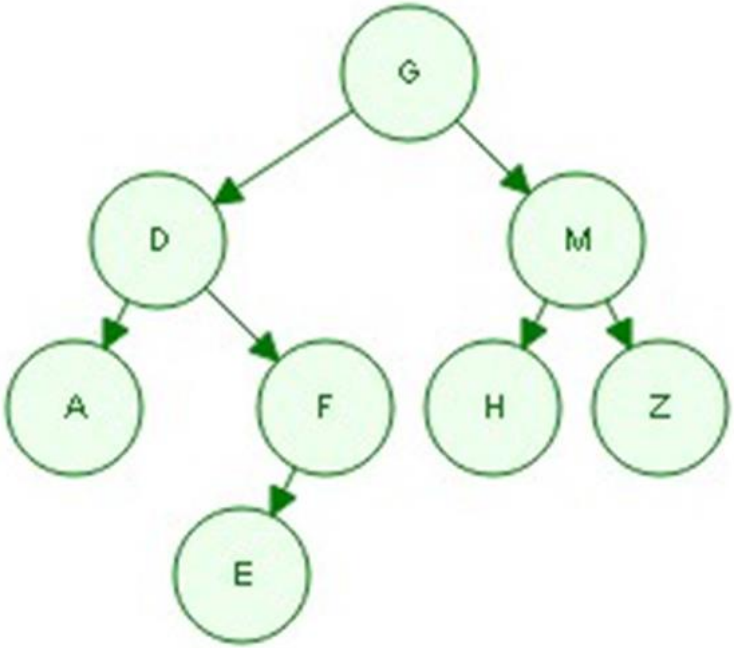
- 如：已知某二叉树的遍历结果如下，求它的后序遍历序列
  - 前序遍历：GDAFEMHZ
  - 中序遍历：ADEFGHMZ
- 两个步骤：
  - 根据前序中序，构造二叉树
  - 后序遍历二叉树

根据前序中序，计算后序

- 前序遍历：GDAFEMHZ
- 中序遍历：ADEFGHMZ
- 根据前序遍历的特点得知，根结点为G；
- 根结点将中序遍历结果ADEFGHMZ分成ADEF和HMZ两个左子树、右子树。
- 递归确定中序遍历序列ADEF和前序遍历序列DAEF的子树结构；
- 递归确定中序遍历序列HMZ和前序遍历序列MHZ的子树结构；

根据前序中序，构造二叉树

- 前序遍历：GDAFEMHZ
- 中序遍历：ADEFGHMZ



Code

```
void InPre2Post(const char* pInOrder, const char* pPreOrder, int nLength, char* pPostOrder, int& nIndex)
{
    if(nLength <= 0)
        return;
    if(nLength == 1)
    {
        pPostOrder[nIndex] = *pPreOrder;
        nIndex++;
        return;
    }
    char root = *pPreOrder;
    int nRoot = 0;
    for(;nRoot < nLength; nRoot++)
    {
        if(pInOrder[nRoot] == root)
            break;
    }
    InPre2Post(pInOrder, pPreOrder+1, nRoot, pPostOrder, nIndex);
    InPre2Post(pInOrder+nRoot+1, pPreOrder+nRoot+1, nLength-(nRoot+1), pPostOrder, nIndex);
    pPostOrder[nIndex] = root;
    nIndex++;
}
```

```
int _tmain(int argc, _TCHAR* argv[])
{
    char pPreOrder[] = "GDAFEMHZ";
    char pInOrder[] = "ADEFGHMZ";
    int size = sizeof(pInOrder) / sizeof(char);
    char* pPostOrder = new char[size];
    int nIndex = 0;
    InPre2Post(pInOrder, pPreOrder, size-1, pPostOrder, nIndex);
    pPostOrder[size-1] = 0;
    cout << pPostOrder << endl;
    delete[] pPostOrder;
    return 0;
}
```

思考

- 若已知二叉树的中序和后序遍历序列，如何求二叉树、如何求二叉树的前序遍历序列呢？

根据中序后序遍历，求前序遍历

- 中序遍历：ADEFGHMZ
- 后序遍历：AEFDHZMG
  - 提示：后序遍历最后一个结点即为根结点，即根结点为G
  - 递归

Code

```
void InPost2Pre(const char* pInOrder, const char* pPostOrder, int nLength, char* pPreOrder, int& nIndex)
{
    if(nLength <= 0)
        return;
    if(nLength == 1)
    {
        pPreOrder[nIndex] = *pPostOrder;
        nIndex++;
        return;
    }
    char root = pPostOrder[nLength-1];
    pPreOrder[nIndex] = root;
    nIndex++;
    int nRoot = 0;
    for(;nRoot < nLength; nRoot++)
    {
        if(pInOrder[nRoot] == root)
            break;
    }
    InPost2Pre(pInOrder, pPostOrder, nRoot, pPreOrder, nIndex);
    InPost2Pre(pInOrder+nRoot+1, pPostOrder+nRoot, nLength-(nRoot+1), pPreOrder, nIndex);
}
```

```
int _tmain(int argc, _TCHAR* argv[])
{
    char pPostOrder[] = "AEFDHZMG";
    char pInOrder[] = "ADEFGHMZ";
    int size = sizeof(pInOrder) / sizeof(char);
    char* pPreOrder = new char[size];
    int nIndex = 0;
    InPost2Pre(pInOrder, pPostOrder, size-1, pPreOrder, nIndex);
    pPreOrder[size-1] = 0;
    cout << pPreOrder << endl;
    delete[] pPreOrder;
    return 0;
}
```

思考

- 给定整数数组，判断该数组有无可能是一颗二叉查找树后序遍历的结果。假定数组中没有重复元素。
  - 编程如何实现？
- 以下序列中不可能是一棵二叉查找树的后序遍历结果的是_____。
  - 1,2,3,4,5
  - 3,5,1,4,2
  - 1,2,5,4,3
  - 5,4,3,2,1

算法描述

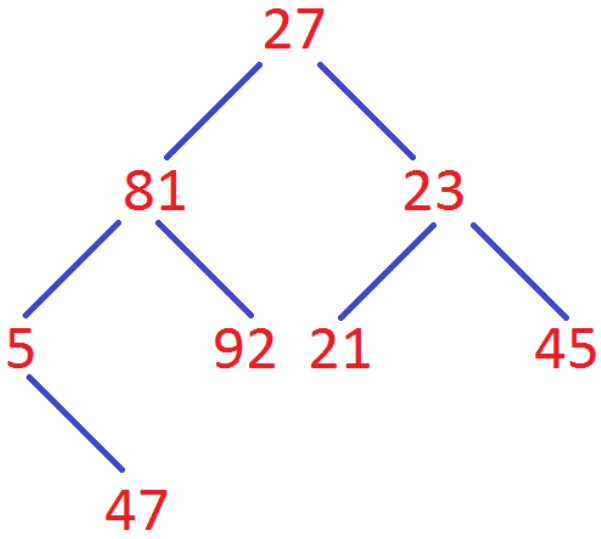
- 1,2,5,4,3
- 由于后序遍历的最后一个元素为根结点，根据该结点，将数组分成前后两段，使得前半段都小于根结点，后半段都大于根结点；如果不存在这样的划分，则不可能是后序遍历的结果。
- 递归判断前后半段是否满足同样的要求。

Code

```
bool CanPostOrder(const int* a, int size)
{
    if(size <= 1)
        return true;

    int root = a[size-1];
    int nLeft = 0;
    while(nLeft < size-1)
    {
        if(a[nLeft] > root)
            break;
        nLeft++;
    }
    int nRight = size-2;    //size-1是根
    while(nRight >= 0)
    {
        if(a[nRight] < root)
            break;
        nRight--;
    }
    if(nRight != nLeft-1)   //无法根据root分成两部分
        return false;

    return CanPostOrder(a, nLeft)           //左子树
    && CanPostOrder(a+nLeft, size-nLeft-1); //右子树
}

int _tmain(int argc, _TCHAR* argv[])
{
    int a[] = {1,2,5,4,3};
    bool b = CanPostOrder(a, sizeof(a)/sizeof(int));
    cout << b << endl;
    return 0;
}
```

Largest BST Subtree

- 给定某二叉树，计算它的最大二叉搜索子树。返回该最大二叉搜索子树的根结点。
- 规定：
  - 如果某子树拥有更多的结点，则该子树更大。
  - 一颗树的子树，指以某结点为根的所有结点。
- 如右图的二叉树，则返回81



题目解析

- 若某结点的左右子树都是二叉搜索树，且能够计算该结点左子树的最大值max和右子树的最小值min，记该结点的值为value。
- 若value > max且value < min，则该结点形成了更大的二叉搜索树。

- 思考：如何代码实现？

Code

```
int _tmain(int argc, _TCHAR* argv[])
{
    //构造输入数据
    CBinaryTree tree;
    for(int i = 0; i < 10; i++)
        tree.Insert(rand() % 100);
    tree.InOrder(ChangeValue);

    tree.InOrder(PrintValue);    //中序遍历
    tree.PreOrder(PrintValue);   //前序遍历

    //计算
    STreeNode* pNode;
    int nLargestNumber = tree.LargestBST(pNode);
    cout << pNode->value << '\t' << nLargestNumber << endl;
    return 0;
}
```

```
void PrintValue(STreeNode* pNode)
{
    cout << pNode->value << '\t';
}

void ChangeValue(STreeNode* pNode)
{
    pNode->value = rand() % 100;
}
```

Code

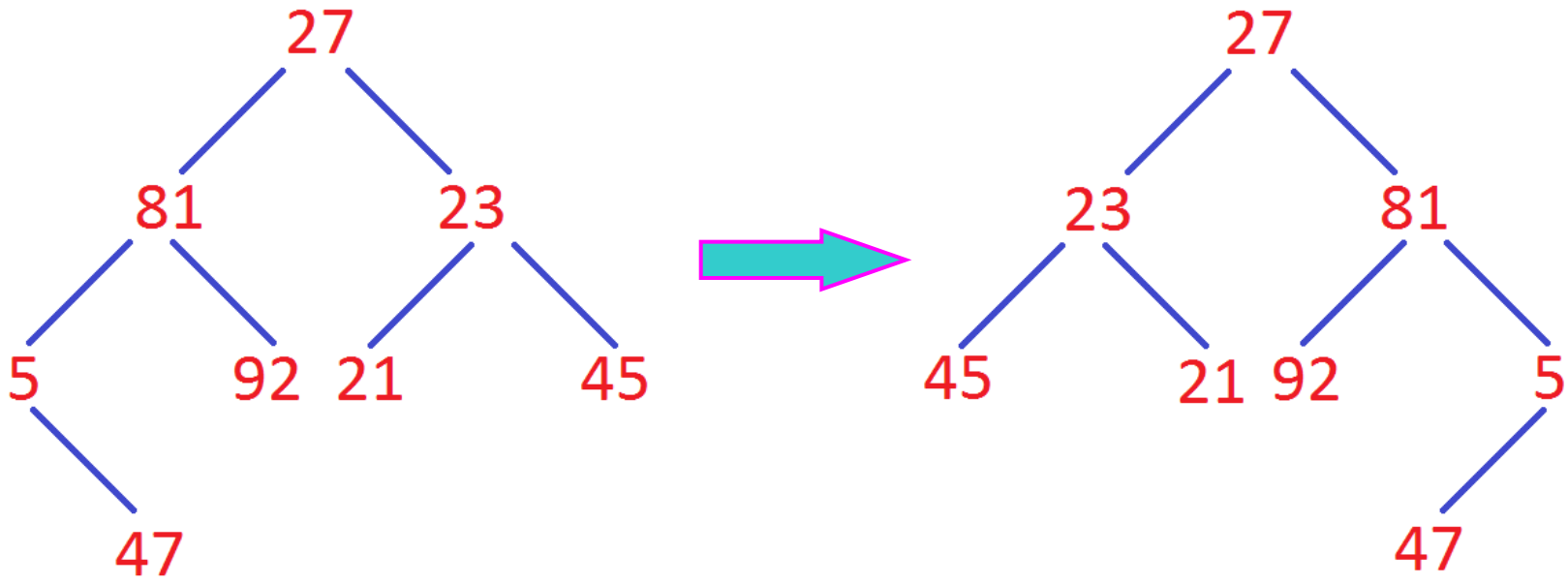
```
int CBinaryTree::LargestBST(STreeNode*& pNode) const
{
    int nMin, nMax, count;
    int nNumber = 0;
    _LargestBST(m_pRoot, nMin, nMax, count, nNumber, pNode);
    return nNumber;
}
```

```
bool CBinaryTree::_LargestBST(STreeNode* pRoot, int& nMin, int& nMax,
    int& count, int& nNumber, STreeNode*& pNode) const
{
    count = 0;
    if(!pRoot)
        return true;
    int nMin1 = INT_MAX, nMin2 = INT_MAX;
    int nMax1 = INT_MIN, nMax2 = INT_MIN;
    int c1, c2;
    if(!_LargestBST(pRoot->pLeft, nMin1, nMax1, c1, nNumber, pNode))
        return false;
    if(!_LargestBST(pRoot->pRight, nMin2, nMax2, c2, nNumber, pNode))
        return false;
    if((pRoot->value < nMax1) || (pRoot->value > nMin2))
        return false;
    count = c1 + c2 + 1;
    nMin = min(nMin1, pRoot->value);
    nMax = max(nMax2, pRoot->value);
    if(count > nNumber)
    {
        nNumber = count;
        pNode = pRoot;
    }
    return true;
}
```

5	45	81	27	61	91	95	42	27	36
27	81	5	45	42	61	91	95	27	36
81	3								

二叉树的翻转

- 给定一颗二叉树，将其左右翻转。
- 如



问题思考

- 对于二叉树的某结点，交换其左右孩子指针然后递归考查左右子树即可。

- 思考：如何代码实现？

Code

```
Before Reverse:
0   24  34  41  58  67  69  78
41  34  0   24  67  58  69  78
After Reverse:
78  69  67  58  41  34  24  0
41  67  69  78  58  34  0   24
```

```
int _tmain(int argc, _TCHAR* argv[])
{
    //构造输入数据
    CBinaryTree tree;
    for(int i = 0; i < 8; i++)
        tree.Insert(rand() % 100);

    tree.InOrder(PrintValue);    //中序遍历
    tree.PreOrder(PrintValue);   //前序遍历

    tree.Reverse();              //翻转
    tree.InOrder(PrintValue);
    tree.PreOrder(PrintValue);
    return 0;
}
```

```
void CBinaryTree::Reverse()
{
    _Reverse(m_pRoot);
}
```

```
void CBinaryTree::_Reverse(STreeNode* pRoot)
{
    if(pRoot)
    {
        swap(pRoot->pLeft, pRoot->pRight);
        _Reverse(pRoot->pLeft);
        _Reverse(pRoot->pRight);
    }
}
```

进一步思考：Code

```
Before Reverse:
0   24  34  41  58  67  69  78
41  34  0   24  67  58  69  78
After Reverse:
78  69  67  58  41  34  24  0
41  67  69  78  58  34  0   24
```

```
int _tmain(int argc, _TCHAR* argv[])
{
    //构造输入数据
    CBinaryTree tree;
    for(int i = 0; i < 8; i++)
        tree.Insert(rand() % 100);

    tree.InOrder(PrintValue);    //中序遍历
    tree.PreOrder(PrintValue);   //前序遍历

    tree.Reverse();              //翻转
    tree.InOrder(PrintValue);
    tree.PreOrder(PrintValue);
    return 0;
}
```

```
void CBinaryTree::Reverse()
{
    _Reverse(m_pRoot);
}
```

```
void CBinaryTree::_Reverse(STreeNode* pRoot)
{
    if(pRoot)
    {
        swap(pRoot->pLeft, pRoot->pRight);
        _Reverse(pRoot->pRight);
        _Reverse(pRoot->pLeft);
    }
}
```

所有括号匹配的字符串

- N对括号能够得到的有效括号序列有哪些？
- 如N=3时，有效括号串共5个，分别为：
  - 1：()()()
  - 2：()(())
  - 3：(())()
  - 4：(()())
  - 5：((()))

问题分析

- 任何一个括号序列，都可以写成形式A(B)
  - A、B都是若干括号对形成的合法串(可为空串)
  - 若N=0，括号序列为空。
  - 若N=1，括号序列只能是()这一种。
- 算法描述：i∈[0,N-1]
  - 计算i对括号的可行序列A；
  - 计算N-i-1对括号的可行序列B；
  - 组合得到A(B)。
  - 注：加上额外一对括号()，总括号共N对

Code

```
1:  () () () () ()      22:  ((())) () ()
2:  () () () (())       23:  ((())) (())
3:  () () (()) ()       24:  (() () ()) ()
4:  () () (() ())       25:  (() (())) ()
5:  () () ((()))        26:  ((()) ()) ()
6:  () (()) () ()       27:  ((() ())) ()
7:  () (()) (())        28:  (((()))) ()
8:  () (() ()) ()       29:  (() () () ())
9:  () ((())) ()        30:  (() () (()))
10:  () (() () ())      31:  (() (()) ())
11:  () (() (()))       32:  (() (() ()))
12:  () ((()) ())       33:  (() ((())))
13:  () ((() ()))       34:  ((()) () ())
14:  () (((())))        35:  ((()) (()))
15:  (()) () () ()      36:  ((() ()) ())
16:  (()) () (())       37:  (((())) ())
17:  (()) (()) ()       38:  ((() () ()))
18:  (()) (() ())       39:  ((() (())))
19:  (()) ((()))        40:  (((()) ()))
20:  (() ()) () ()      41:  (((() ())))
21:  (() ()) (())       42:  ((((()))))
```

```
void Unit(vector<string>& result,
    const vector<string>& prefix, const vector<string>& suffix)
{
    vector<string>::const_iterator ip,is;
    for(ip = prefix.begin(); ip != prefix.end(); ip++)
    {
        for(is = suffix.begin(); is != suffix.end(); is++)
        {
            result.push_back("");
            string& r = result.back();
            r += "(";
            r += *ip;
            r += ")";
            r += *is;
        }
    }
}

vector<string> AllParentheses(int n)
{
    if(n == 0)
        return vector<string>(1, "");
    if(n == 1)
        return vector<string>(1, "()");
    vector<string> prefix, suffix, result;
    for(int i = 0; i < n; i++)
    {
        prefix = AllParentheses(i);
        suffix = AllParentheses(n-i-1);
        Unit(result, prefix, suffix);
    }
    return result;
}
```

思考

- 本题可以看做隐式树的建立和搜索。
- 可以通过增加缓存的方式，对已经计算得到的字串直接获取，以空间换时间，降低时间复杂度。
- 如果只是计算可行括号串的数目，如何计算？
  - 事实上，数组A[i]表示长度为i的括号串的可行数目，即著名的Catalan数。
  - 该问题在动态规划中继续讨论。
- Calalan数(从0开始数)：
  - 1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, 16796, 58786, 208012, 742900, 2674440, 9694845, 35357670, 129644790, 477638700, 1767263190, 6564120420, 24466267020, 91482563640, 343059613650, 1289904147324, 4861946401452……

Code

```
 0: 1
 1: 1
 2: 2
 3: 5
 4: 14
 5: 42
 6: 132
 7: 429
 8: 1430
 9: 4862
10: 16796
11: 58786
12: 208012
13: 742900
14: 2674440
15: 9694845
16: 35357670
17: 129644790
18: 477638700
19: 1767263190
```

```
void GetCatalan(int* pCatalan, int N)
{
    pCatalan[0] = 1;
    pCatalan[1] = 1;
    int i,j;
    int c;
    for(i = 2; i <= N; i++)
    {
        pCatalan[i] = 1;
        c = 0;
        for(j = 0; j < i; j++)
        {
            c += pCatalan[j] * pCatalan[i-j-1];
        }
        pCatalan[i] = c;
    }
}

int _tmain(int argc, _TCHAR* argv[])
{
    const int n = 19;
    int catalan[n+1];
    GetCatalan(catalan, n);
    PrintNumber(catalan, n);
    return 0;
}
```

K个不同字符的最长子串

- 给定字符串str，计算最多包括k个不同字符的最长子串。
  - 如给定字符串“eceba”和k=3，则包括3个不同字符的最长子串为“eceb”。

算法描述

- 使用双索引i，j，初值赋为0；考察子串str[i,j]
  - 为表述方便，记“子串str[i,j]中包含的不同字符数目小于等于k”为A。
  - 若子串str[i,j]满足A，则j++；
  - 若子串str[i,j]不满足A，则i++，继续考察str[i,j]是否满足A。
- 思考：如何快速考察str[i,j]的字符数目呢？
  - 考察：str[i,j]不断发生改变

Code

```
int LongestSubstringK(const char* str, int size, int k)
{
    if(size <= 0)
        return 0;
    map<char, int> wd;
    int left = 0;
    int mx = 0;
    for(int right = 0; right < size; right++)
    {
        wd[str[right]]++;
        while((int)wd.size() > k)
        {
            wd[str[left]]--;
            if(wd[str[left]] == 0)
                wd.erase(str[left]);
            left++;
        }
        mx = max(right-left+1, mx);
    }
    return mx;
}

int _tmain(int argc, _TCHAR* argv[])
{
    char s[] = "eceba";
    cout << LongestSubstringK(s, sizeof(s)/sizeof(char)-1, 3);
    return 0;
}
```

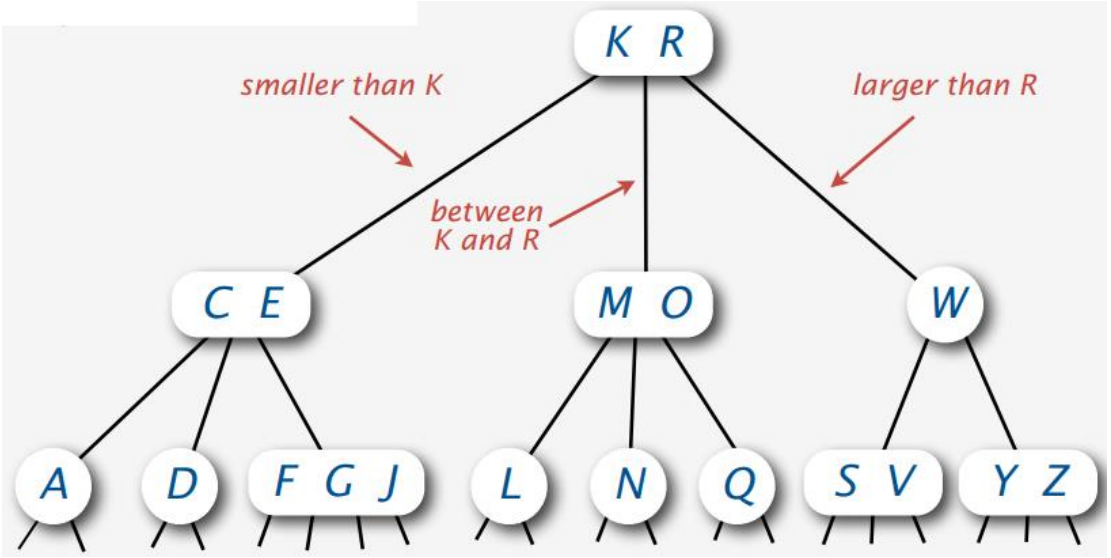
Code2

```
int LongestSubstringK2(const char* str, int size, int k, int& from, int& to)
{
    if(size <= 0)
        return 0;
    map<char, int> wd;
    int left = 0;
    int mx = 0;
    for(int right = 0; right < size; right++)
    {
        wd[str[right]]++;
        while((int)wd.size() > k)
        {
            wd[str[left]]--;
            if(wd[str[left]] == 0)
                wd.erase(str[left]);
            left++;
        }
        if(mx < right-left+1)
        {
            mx = right - left + 1;
            from = left;
            to = right;
        }
    }
    return mx;
}
```

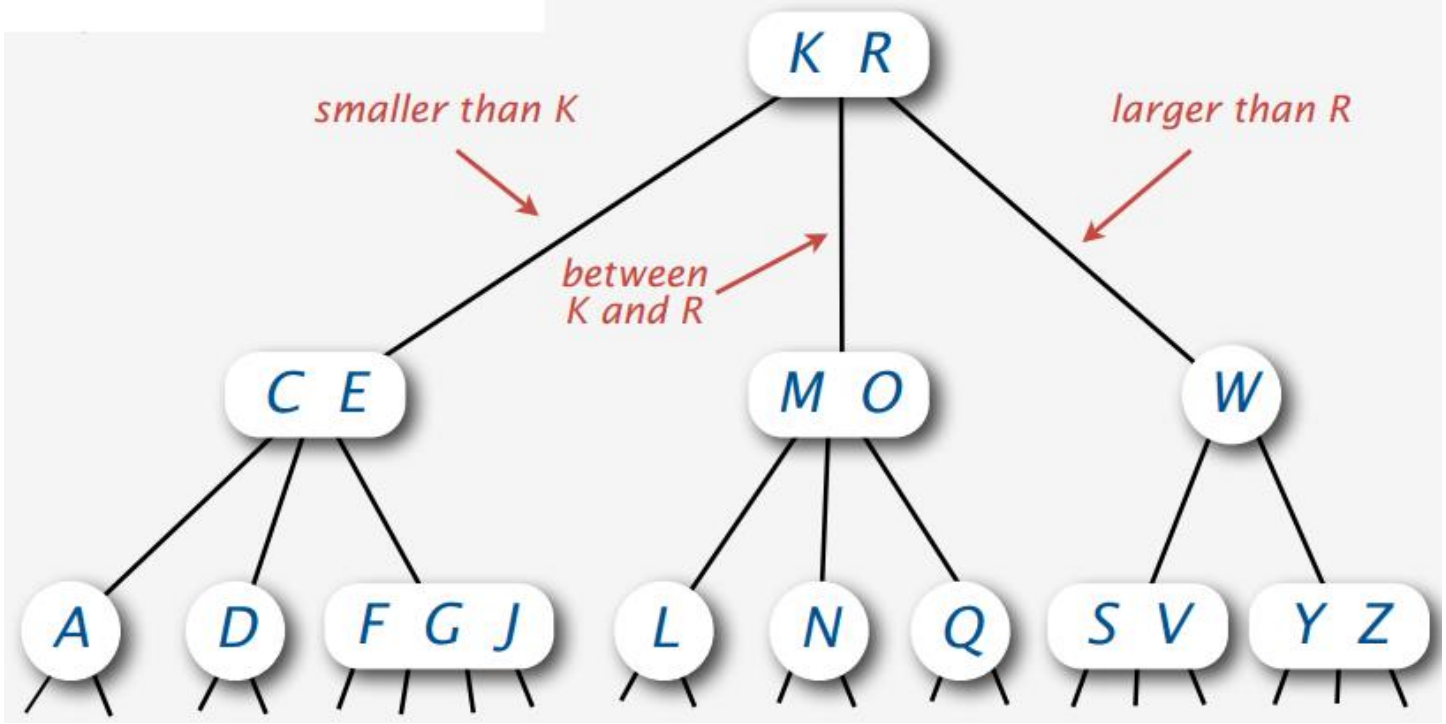
```
int _tmain(int argc, _TCHAR* argv[])
{
    char s[] = "eceba";
    int from, to;
    LongestSubstringK2(s, sizeof(s)/sizeof(char)-1, 3, from, to);
    Print(s, from, to);
    return 0;
}
```

二叉到多叉的思考

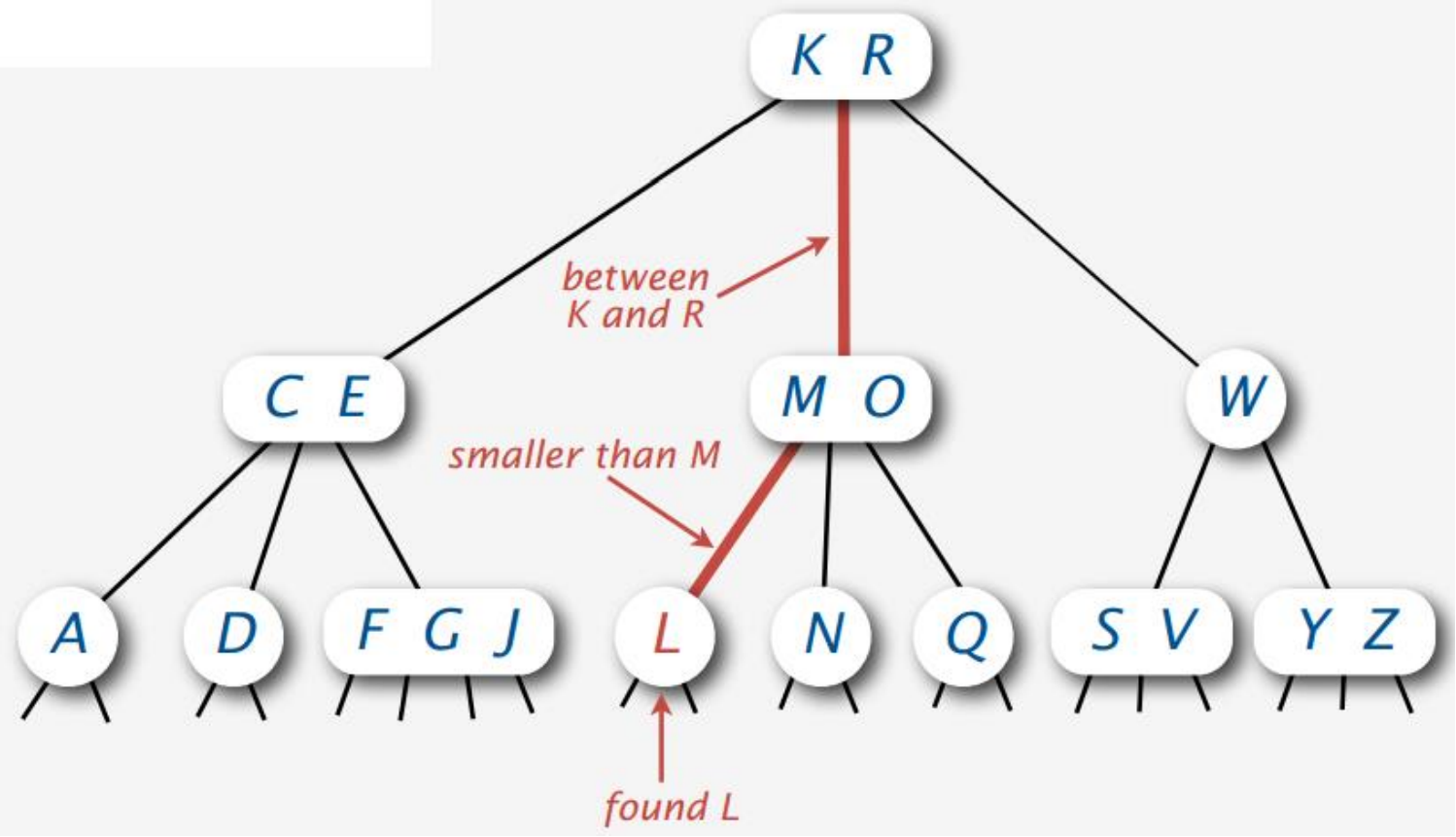
- 一个结点存一个值，则有2个孩子：W
- 一个结点存两个值，则有3个孩子：MO
- 一个结点存三个值，则有4个孩子：FGJ



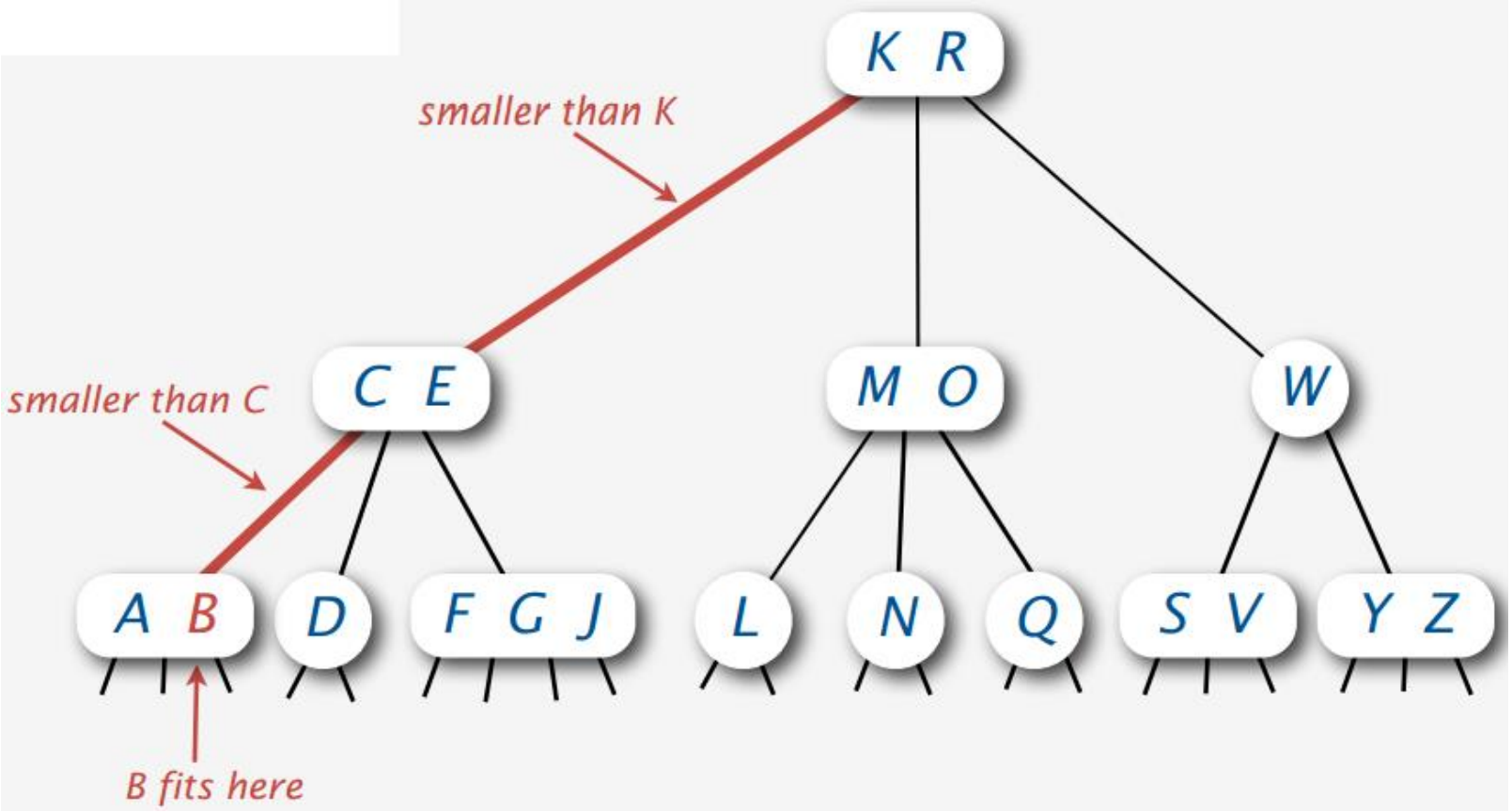
2-3-4树


K R
smaller than K
larger than R
between
K and R
C E
M O
W
A
D
F G J
L
N
Q
S V
Y Z

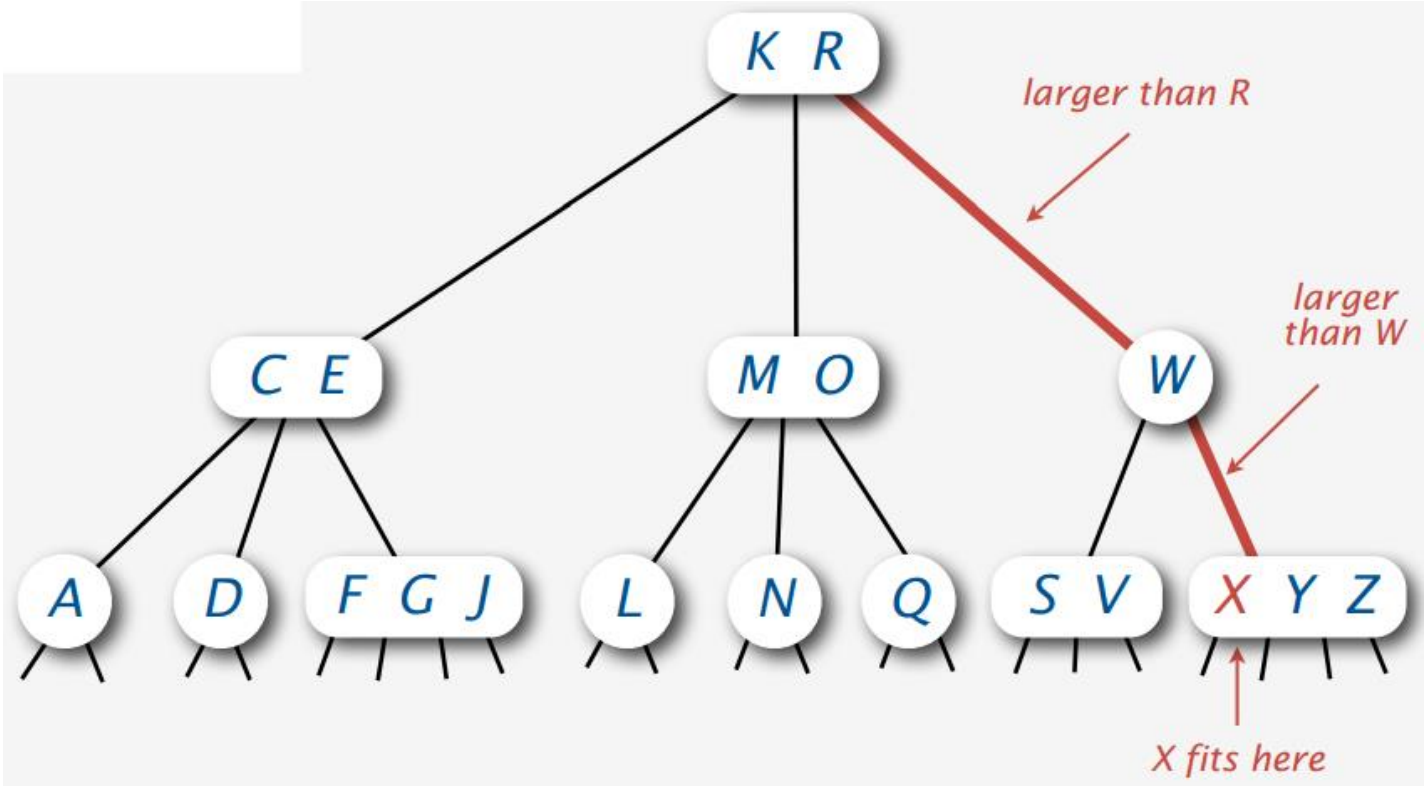
查找L



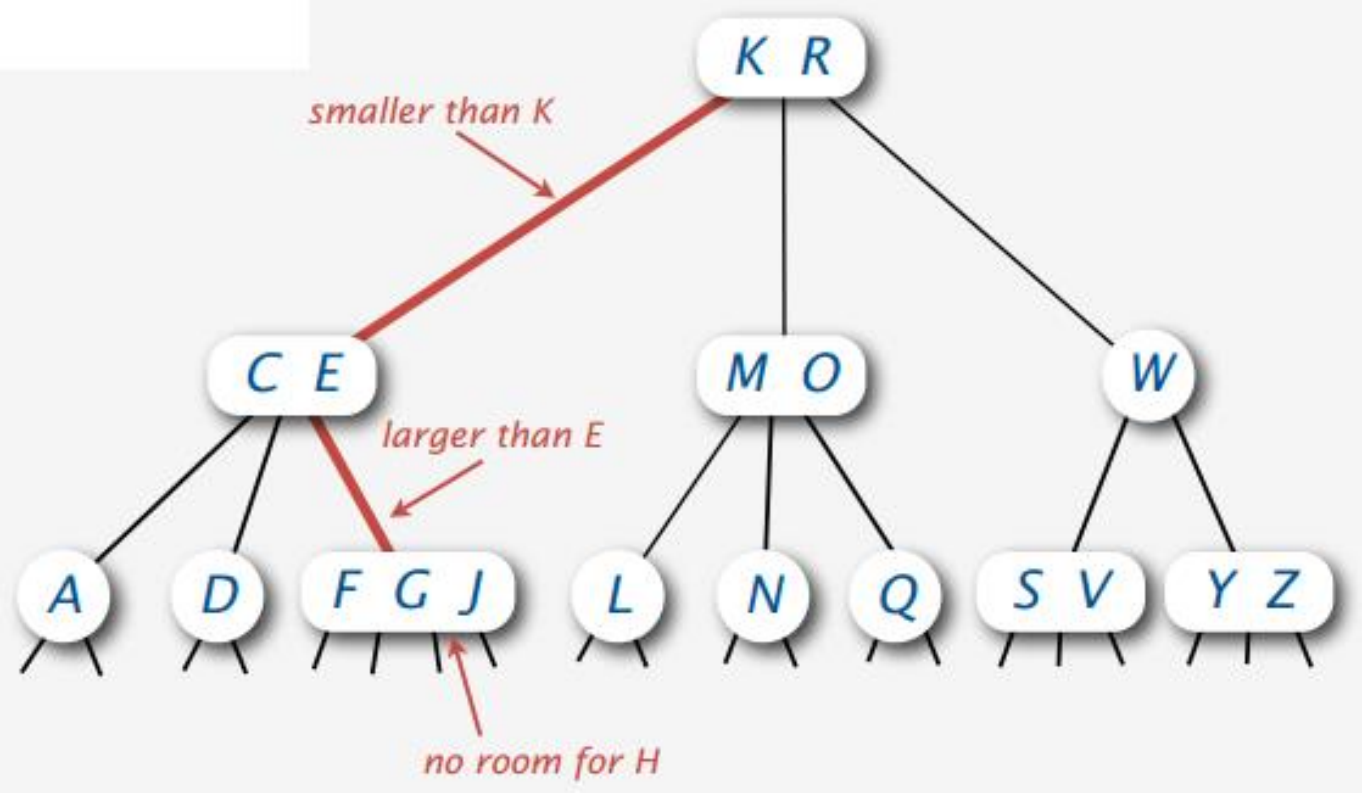
插入B


K R
smaller than K
C E
smaller than C
M O
W
A B
D
F G J
L
N
Q
S V
Y Z
B fits here

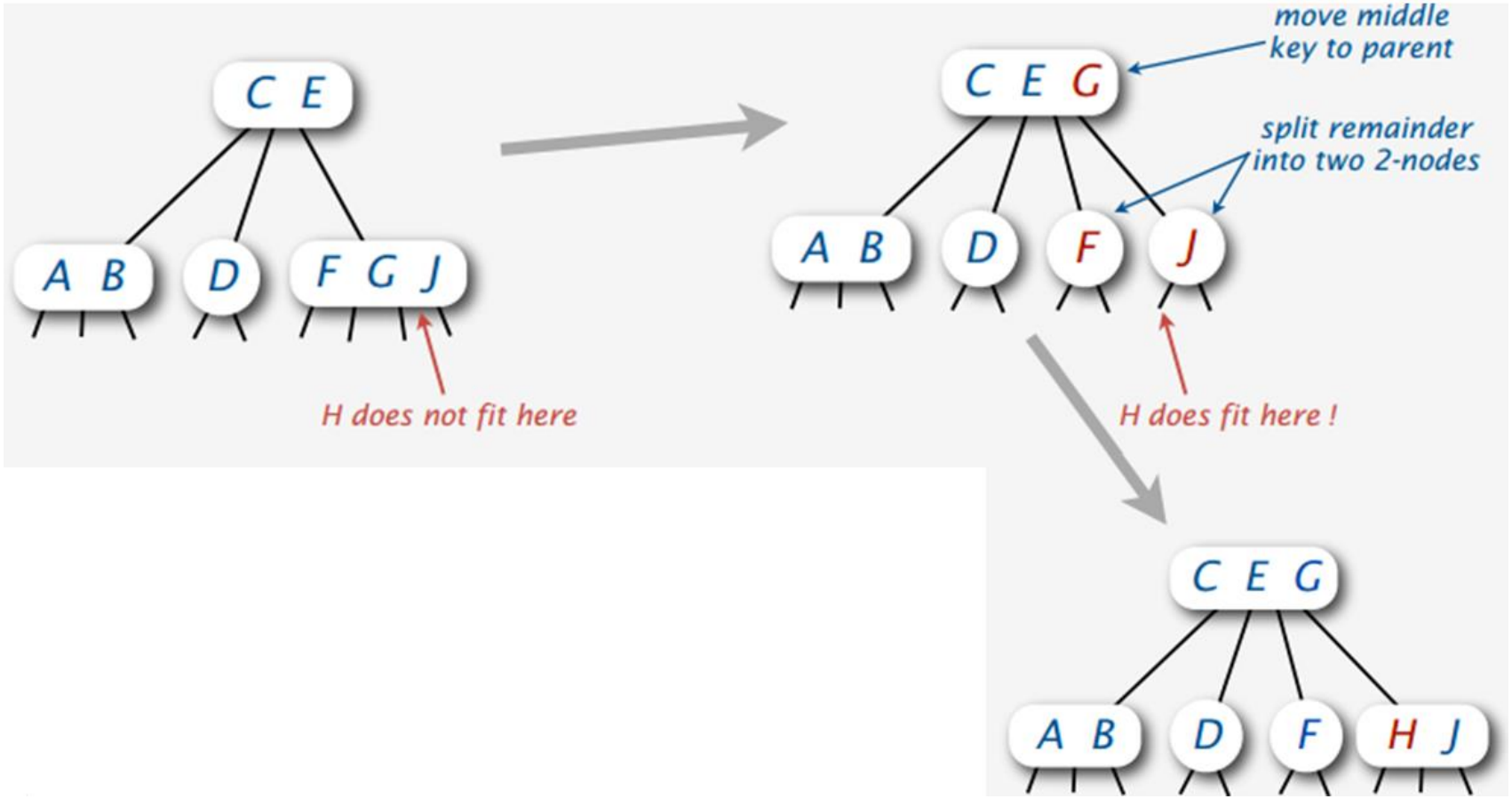
插入X


K R
larger than R
larger
than W
C E
M O
W
A
D
F G J
L
N
Q
S V
X Y Z
X fits here

插入H


K R
smaller than K
C E
larger than E
M O
W
A
D
F G J
L
N
Q
S V
Y Z
no room for H

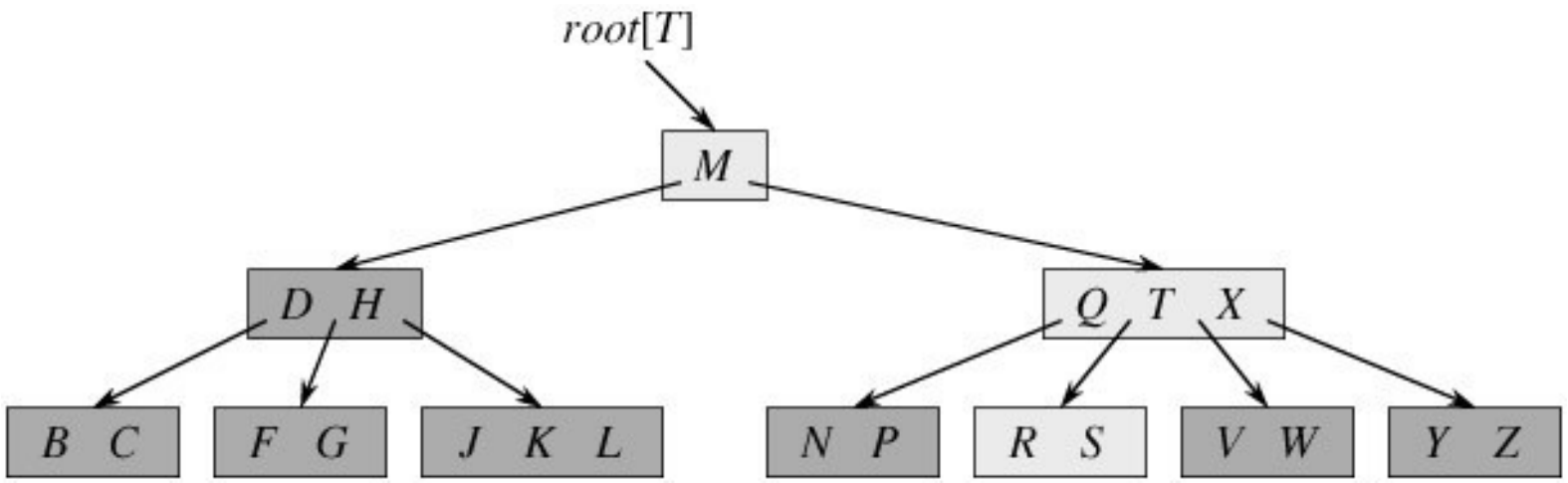
分裂结点


C E
A B
D
F G J
H does not fit here
move middle key to parent
split remainder into two 2-nodes
C E G
A B
D
F
J
H does fit here !
C E G
A B
D
F
H J

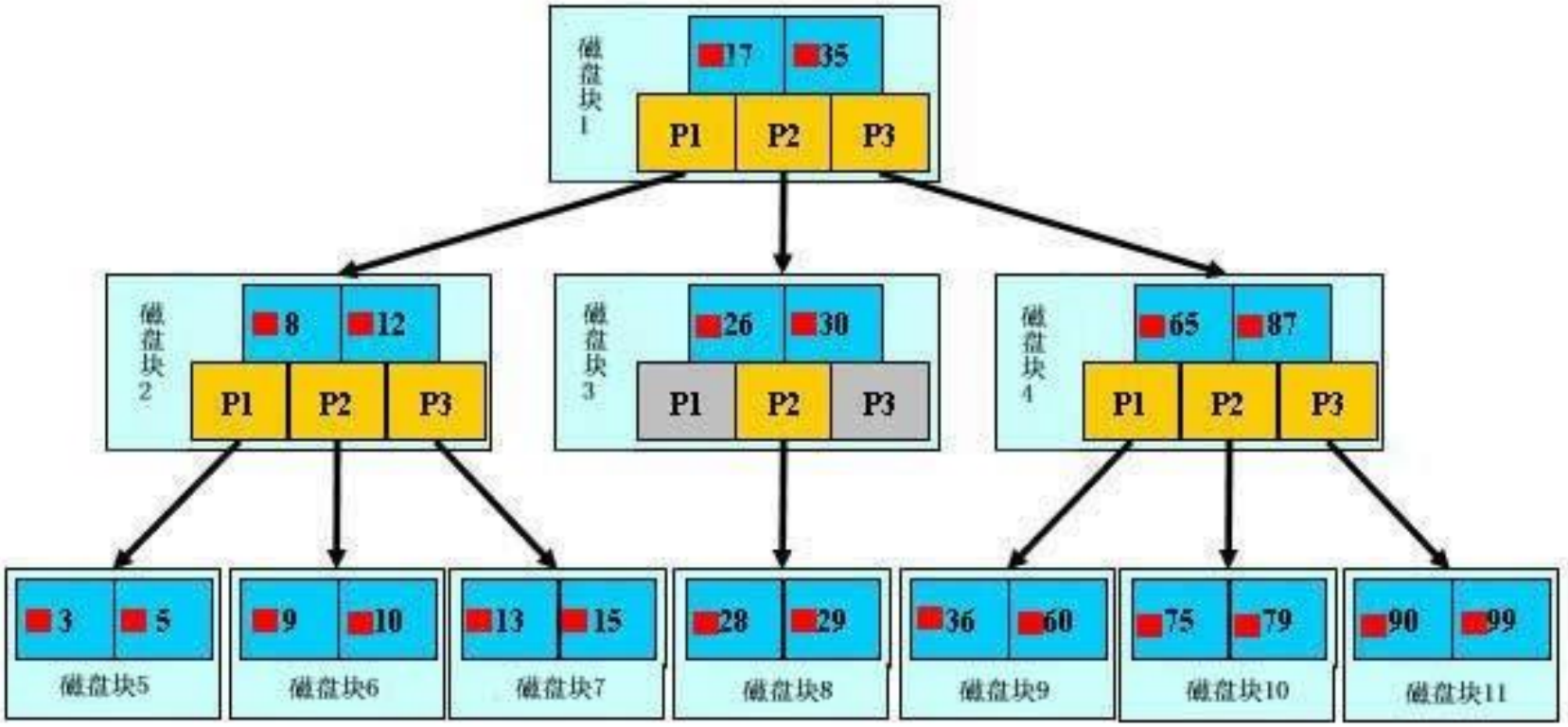
B树的定义

- m阶B树需要满足的条件：
  - 每个结点至多有m个孩子；
  - 除根结点外，其他结点至少有m/2个孩子；
  - 根结点至少有2个孩子；
  - 所有叶结点在同一层；
  - 有α个孩子的非叶结点有α-1个关键字；结点内部，关键字递增排列。

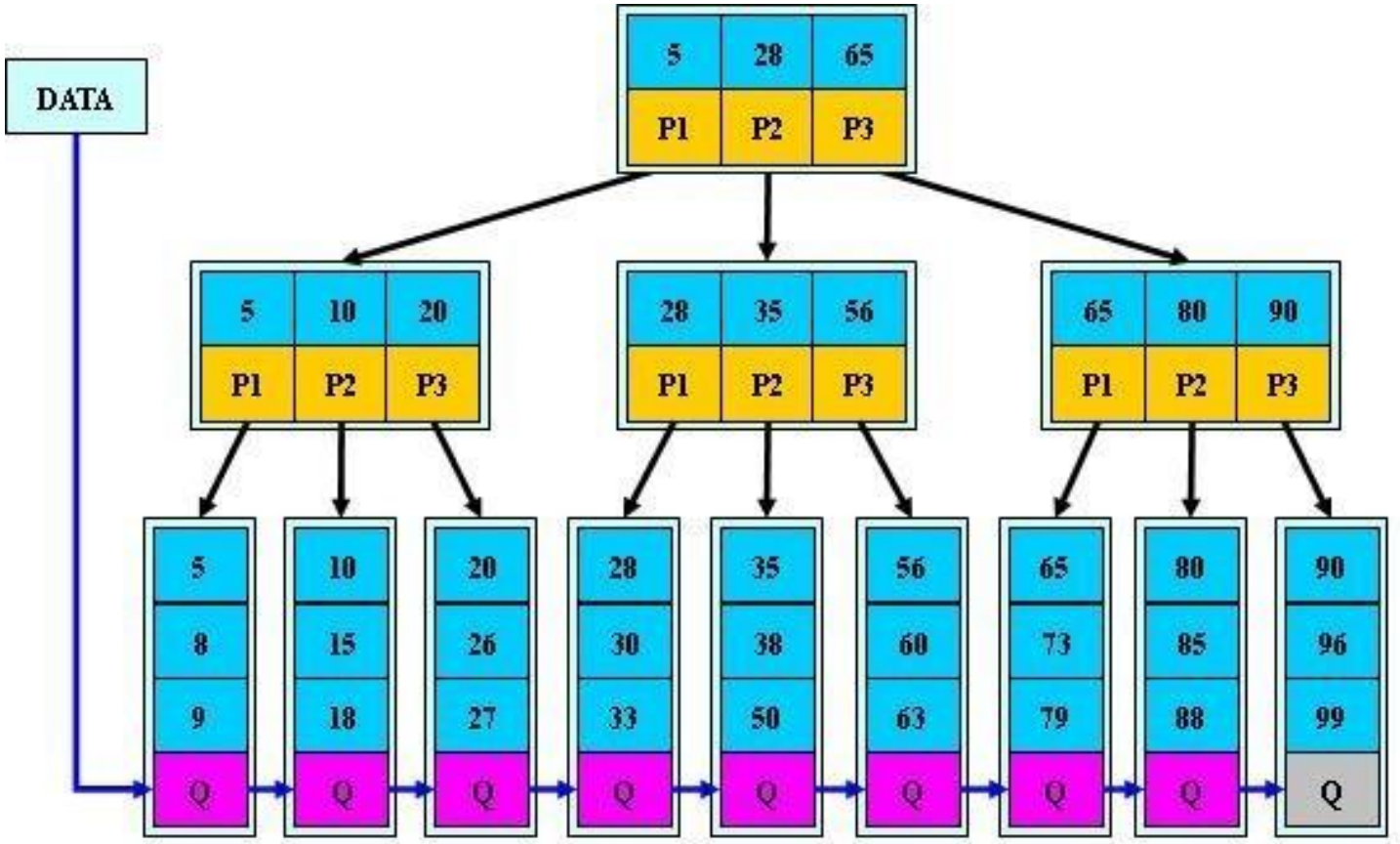
B树



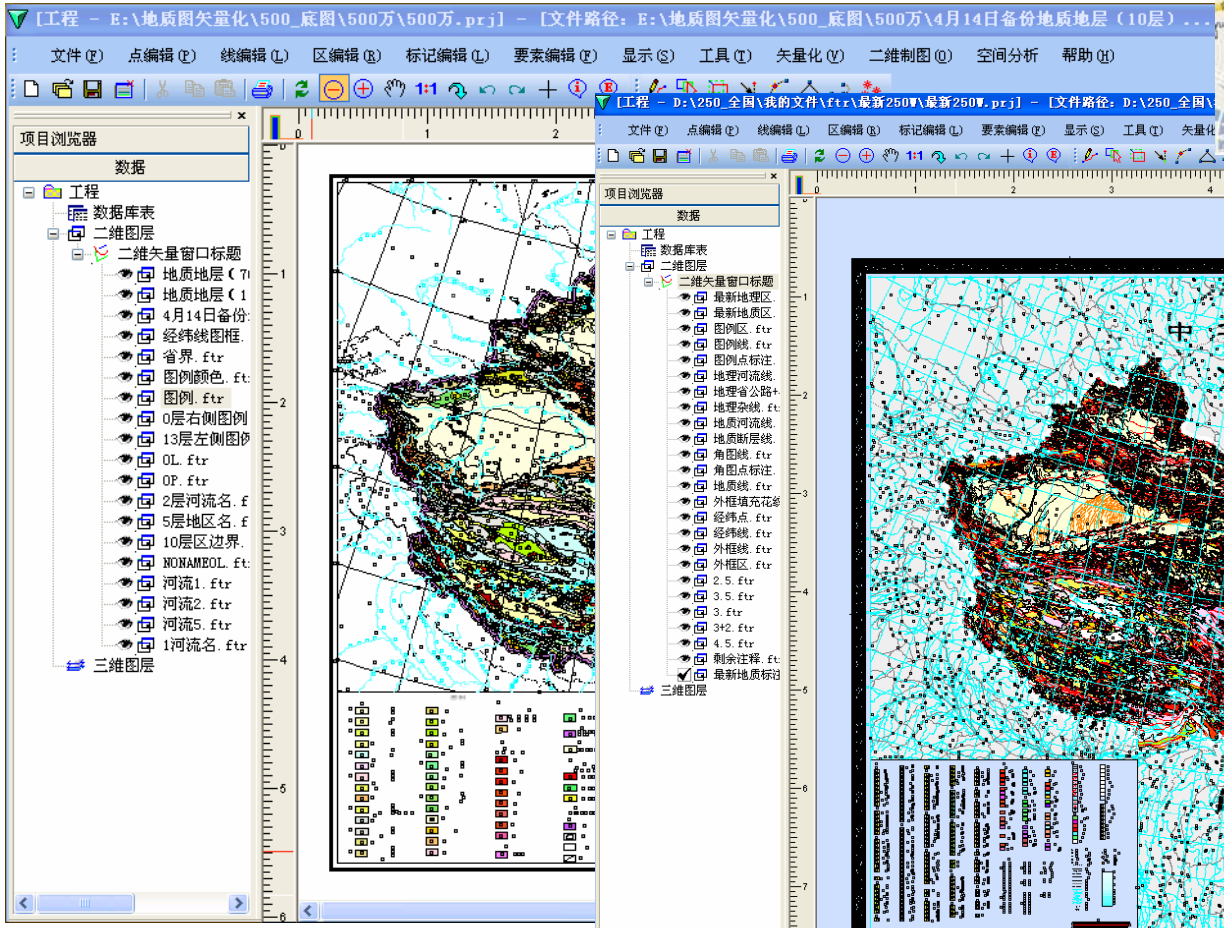
B树

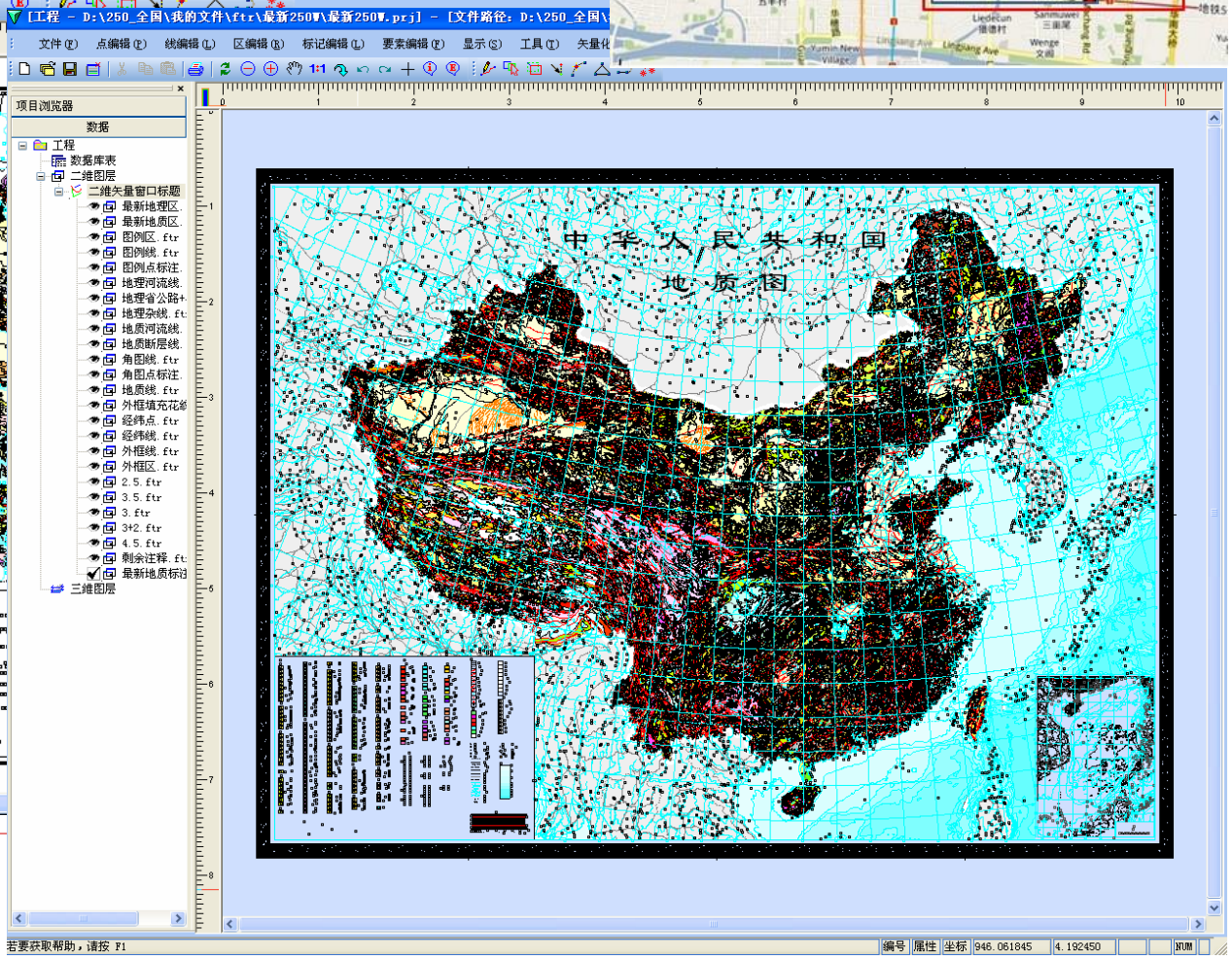

磁盘块1
17
35
P1
P2
P3
磁盘块2
8
12
P1
P2
P3
磁盘块3
26
30
P1
P2
P3
磁盘块4
65
87
P1
P2
P3
3
5
磁盘块5
9
10
磁盘块6
13
15
磁盘块7
28
29
磁盘块8
36
60
磁盘块9
75
79
磁盘块10
90
99
磁盘块11

B树的变种

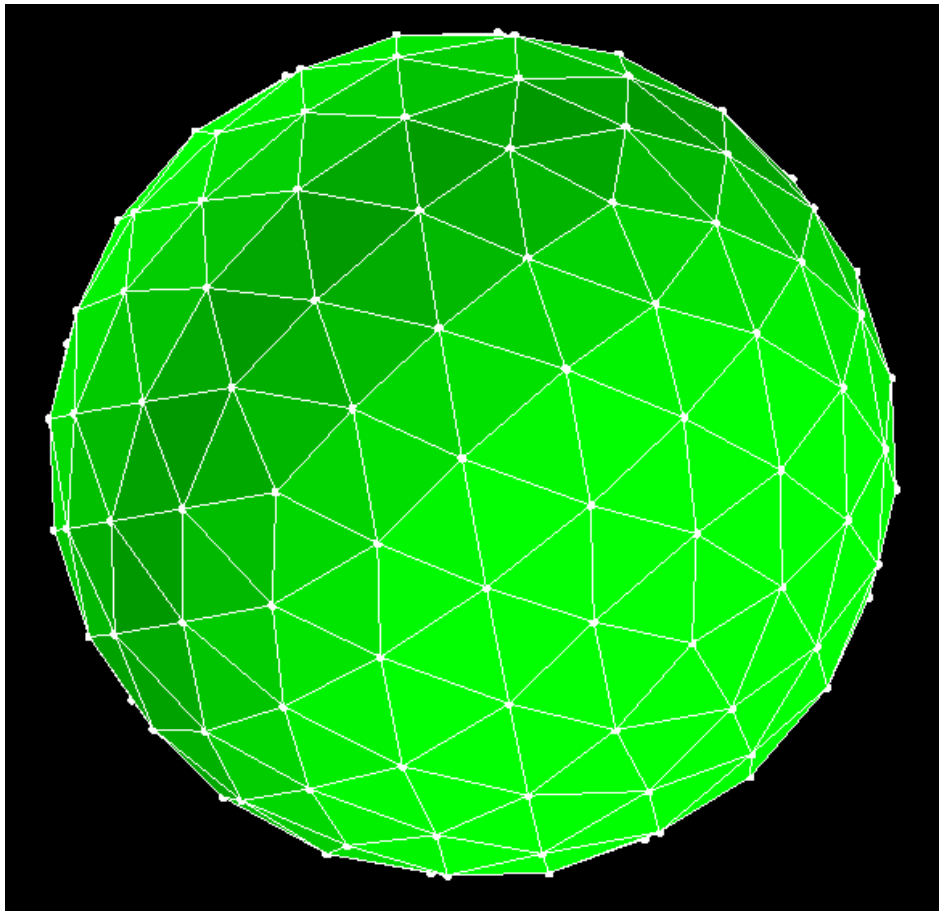

DATA
5
28
65
P1
P2
P3
5
10
20
P1
P2
P3
28
35
56
P1
P2
P3
65
80
90
P1
P2
P3
5
8
9
Q
10
15
18
Q
20
26
27
Q
28
30
33
Q
35
38
50
Q
56
60
63
Q
65
73
79
Q
80
85
88
Q
90
96
99
Q

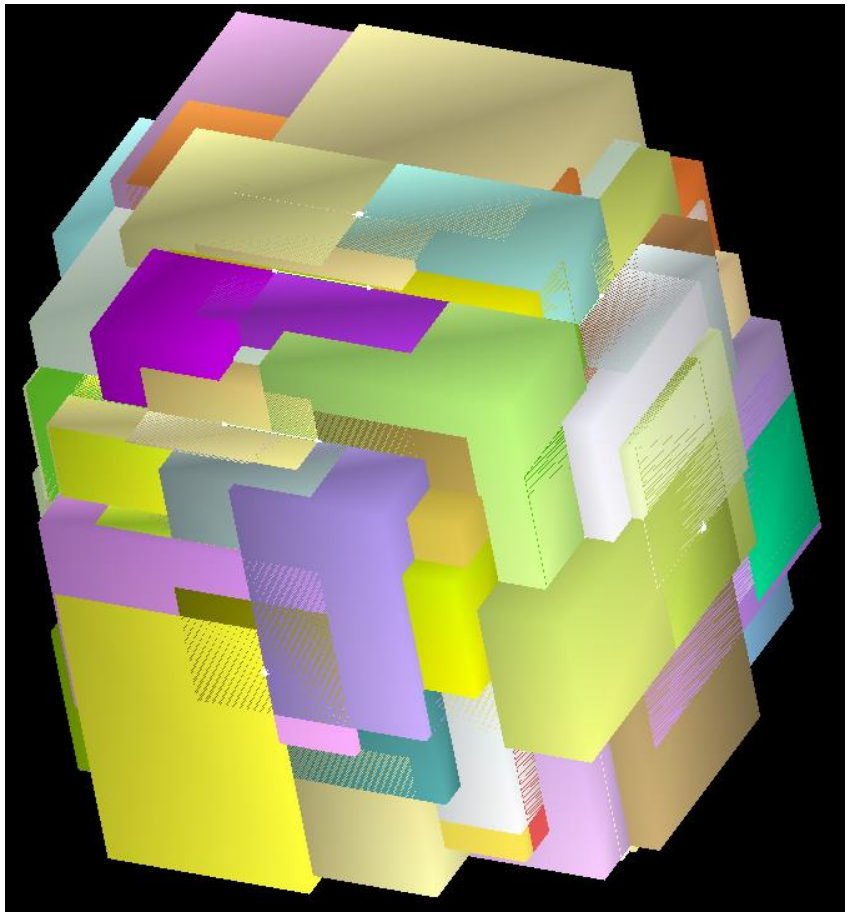
二维上的B树——R树


[工程 - E:\地质图矢量化\500_底图\500万\500万.prj] - [文件路径: E:\地质图矢量化\500_底图\500万\4月14日备份地质地层（10层）...
文件(F) 点编辑(P) 线编辑(L) 区编辑(R) 标记编辑(L) 要素编辑(F) 显示(S) 工具(T) 矢量化(V) 二维制图(Q) 空间分析 帮助(H)
项目浏览器
数据
工程
数据库表
二维图层
二维矢量窗口标题
地质地层(7
地质地层(1
4月14日备份
经纬线图框.
省界.ftr
图例颜色.ft
图例.ftr
0层右侧图例
13层左侧图例
OL.ftr
OP.ftr
2层河流名.f
5层地区名.f
10层区边界.
NONAMEOL.ft
河流1.ftr
河流2.ftr
河流5.ftr
1河流名.ftr
三维图层

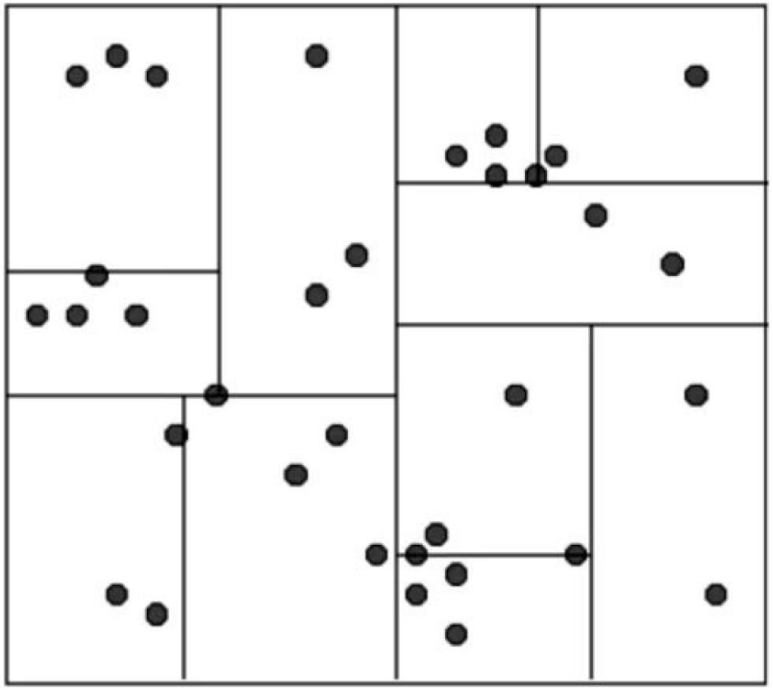

[工程 - D:\250_全国\我的文件\ftr\最新250W\最新250W.prj] - [文件路径: D:\250_全国\
文件(F) 点编辑(P) 线编辑(L) 区编辑(R) 标记编辑(L) 要素编辑(F) 显示(S) 工具(T) 矢量化
项目浏览器
数据
工程
数据库表
二维图层
二维矢量窗口标题
最新地理区.
最新地质区.
图例区.ftr
图例线.ftr
图例点标注.
地理河流线.
地理省公路+
地理杂线.ft
地质河流线.
地质断层线.
角图线.ftr
角图点标注.
地质线.ftr
外框填充花彩
经纬点.ftr
经纬线.ftr
外框线.ftr
外框区.ftr
2.5.ftr
3.5.ftr
3.ftr
3+2.ftr
4.5.ftr
剩余注释.ft
最新地质标注
三维图层
中华人民共和国
地质图
若要获取帮助，请按 F1
编号 属性 坐标 946.061845 4.192450 NUM

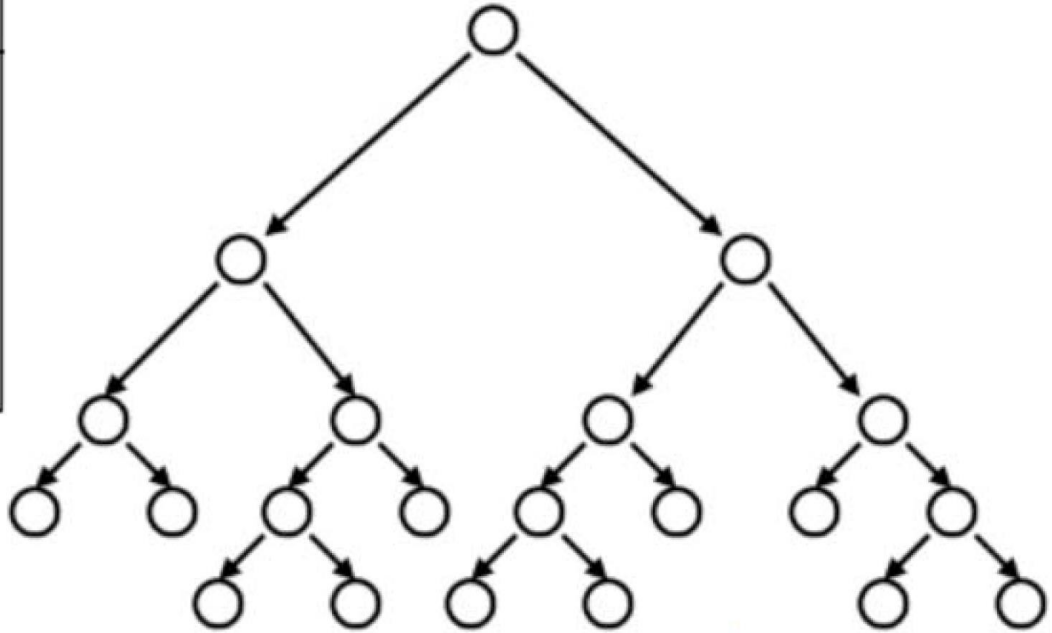
三维空间中的R树



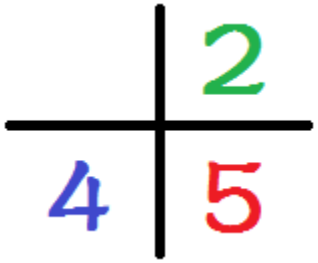


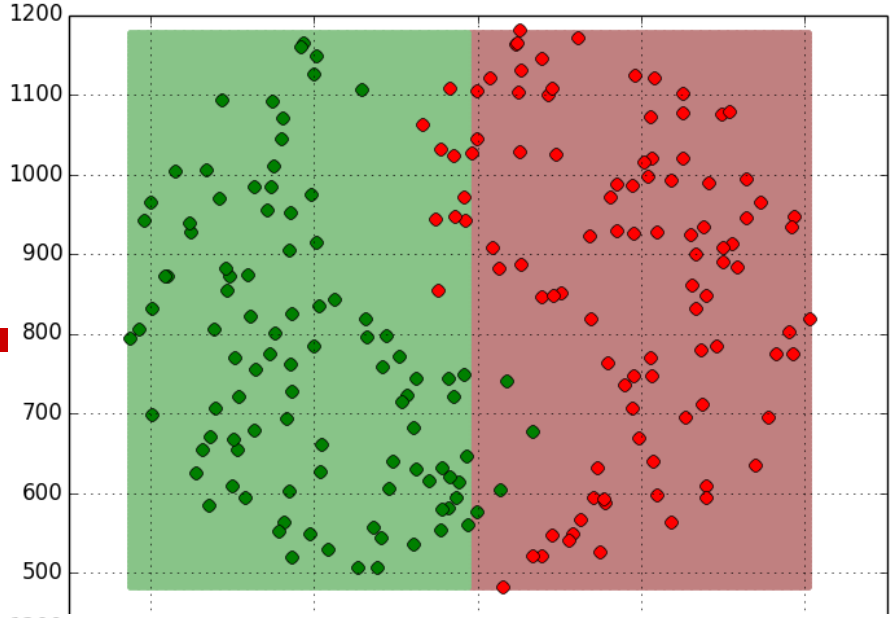
KD-Tree

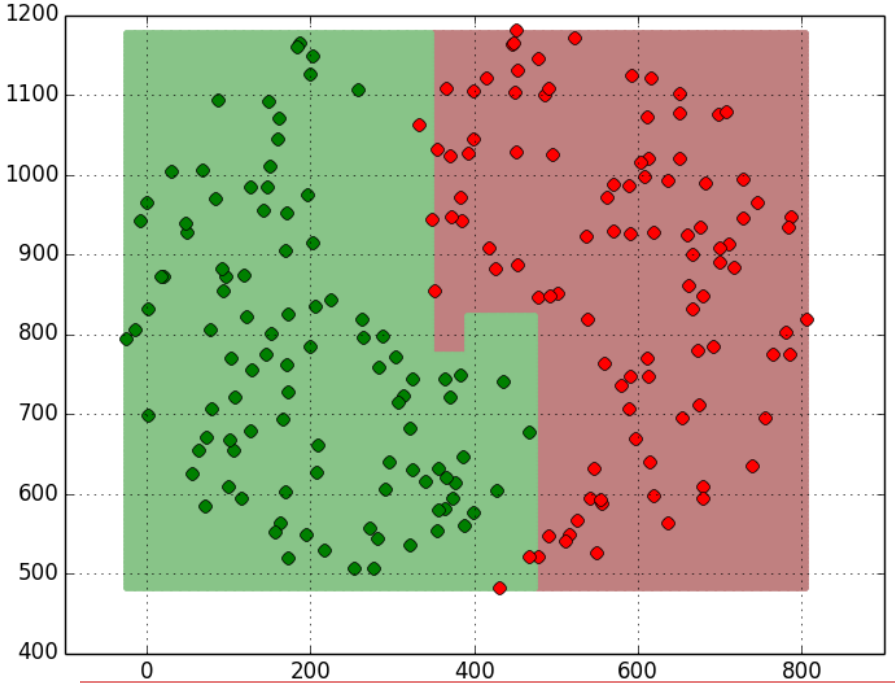


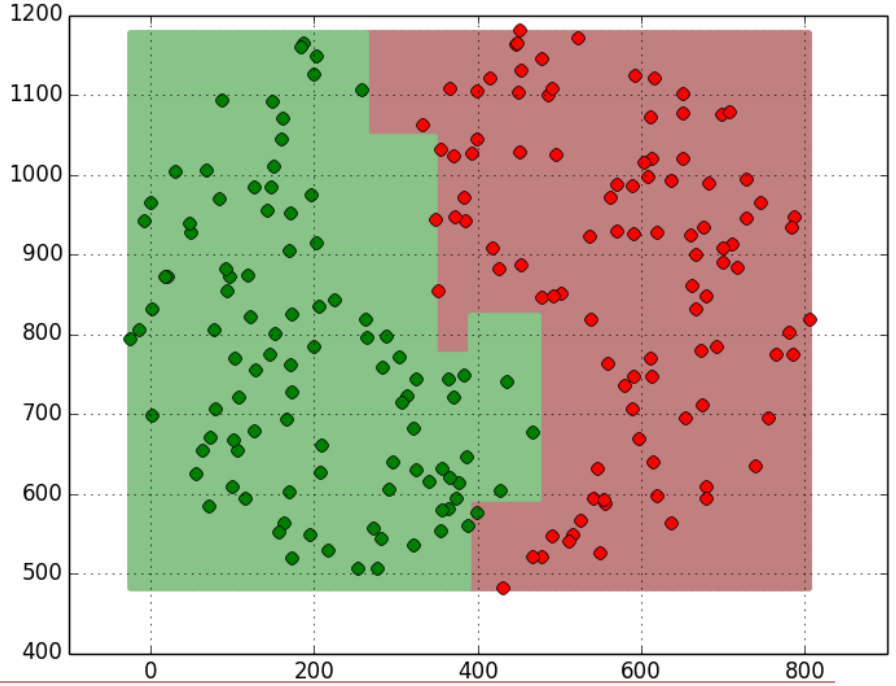


决策树：Level

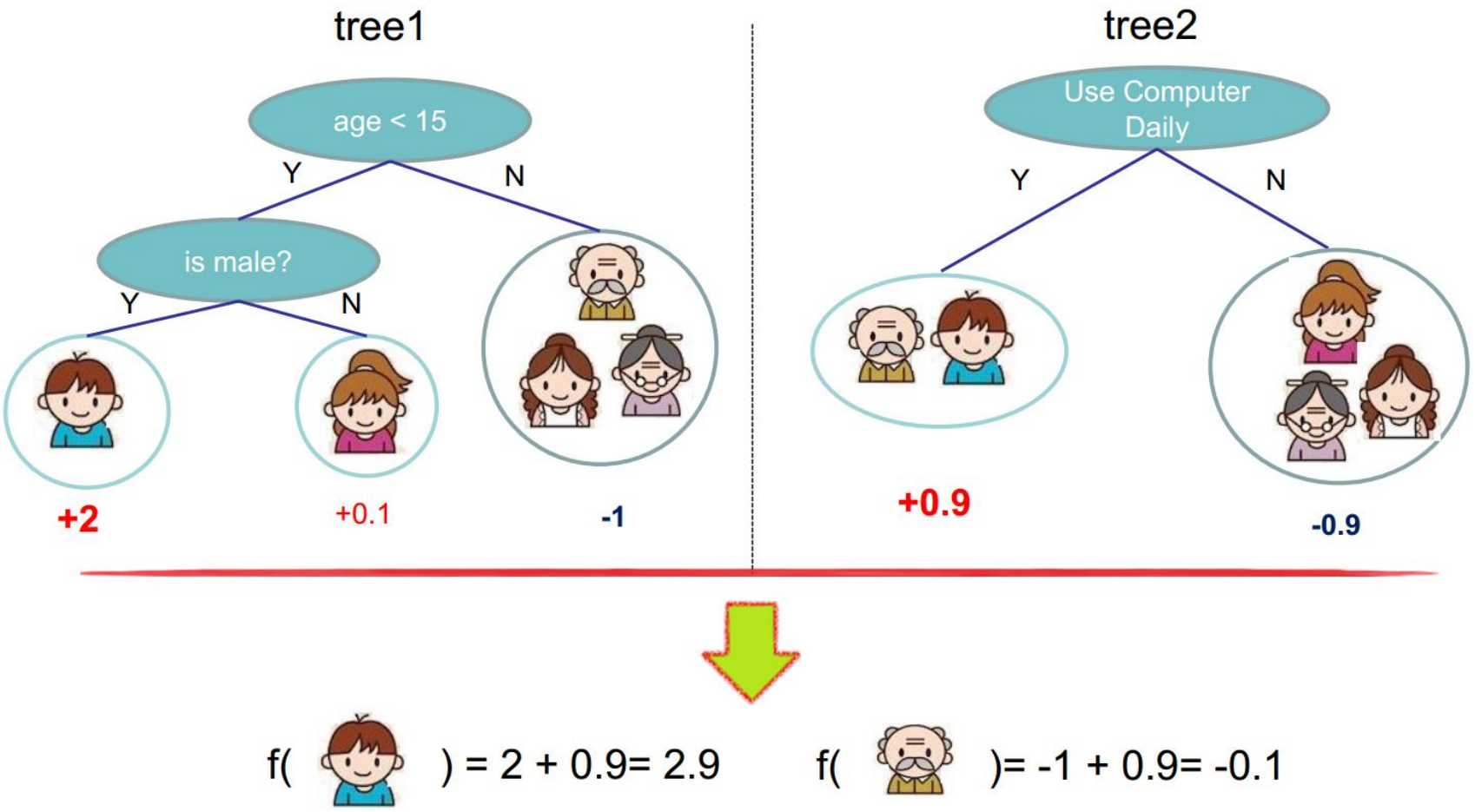

2
4
5


1200
1100
1000
900
800
700
600
500

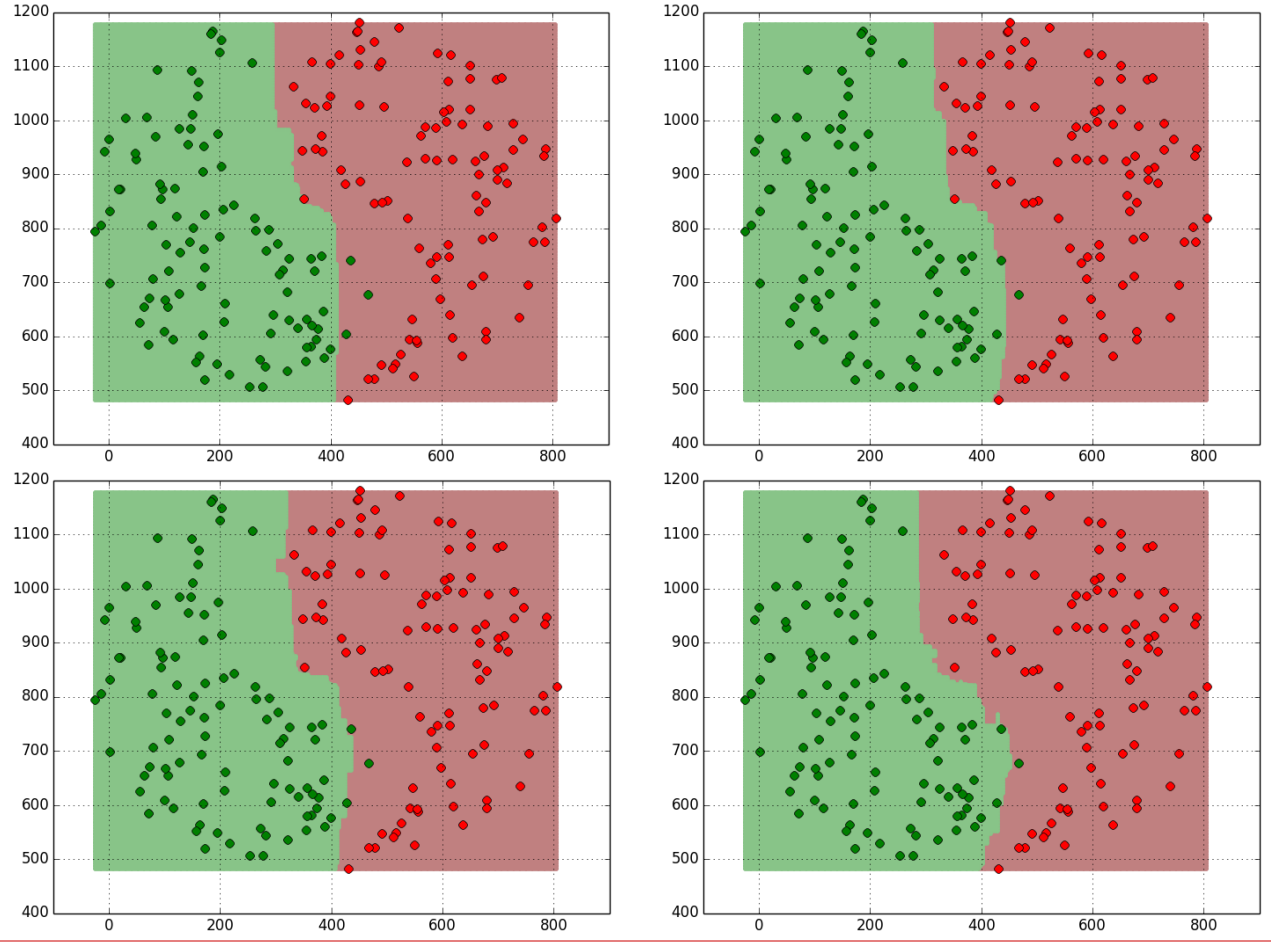

1200
1100
1000
900
800
700
600
500
400
0
200
400
600
800


1200
1100
1000
900
800
700
600
500
400
0
200
400
600
800

随机森林

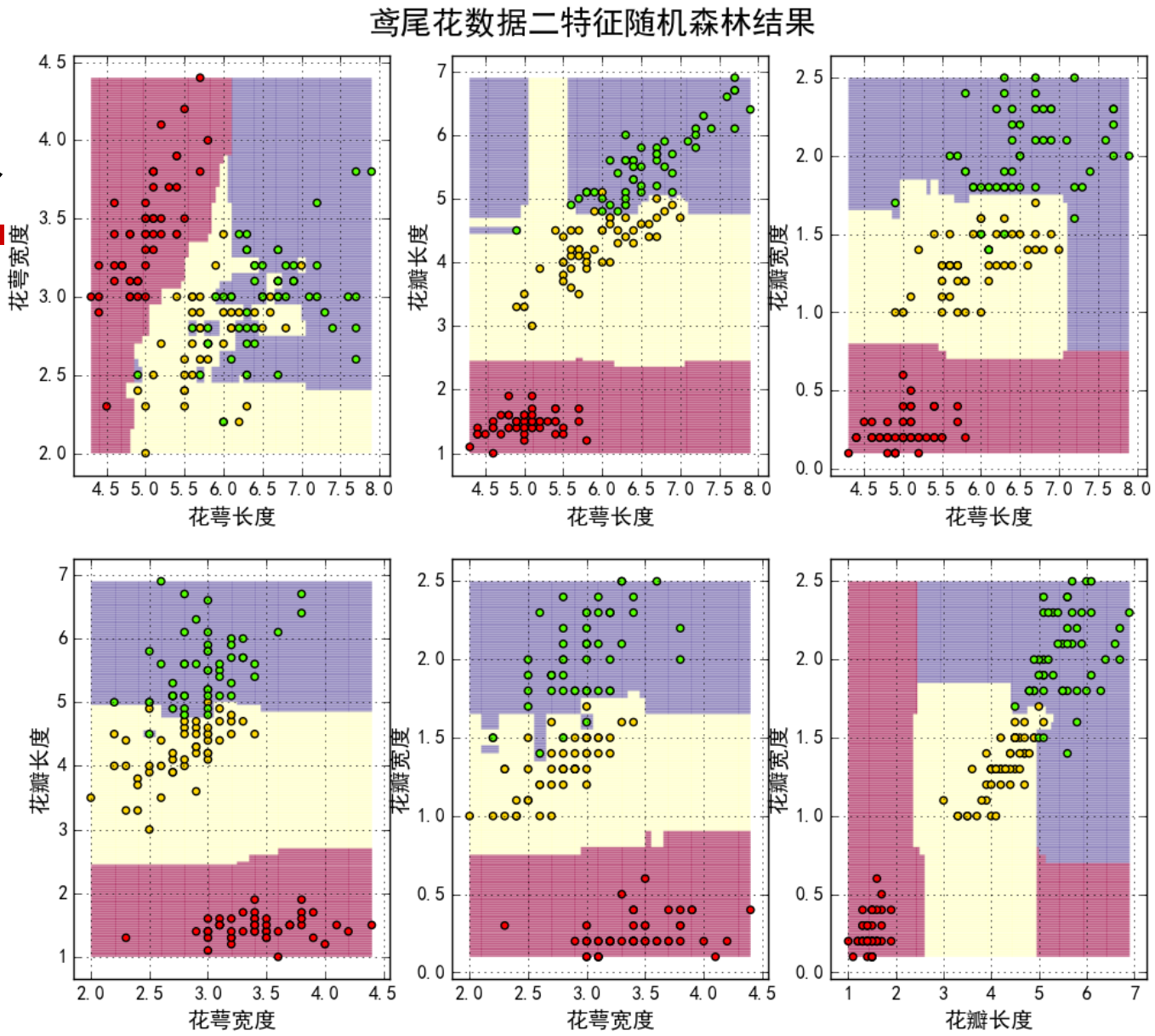

tree1
age < 15
Y
N
is male?
Y
N
+2
+0.1
-1
tree2
Use Computer Daily
Y
N
+0.9
-0.9
f( ) = 2 + 0.9= 2.9
f( )= -1 + 0.9= -0.1

随机森林：30，Level

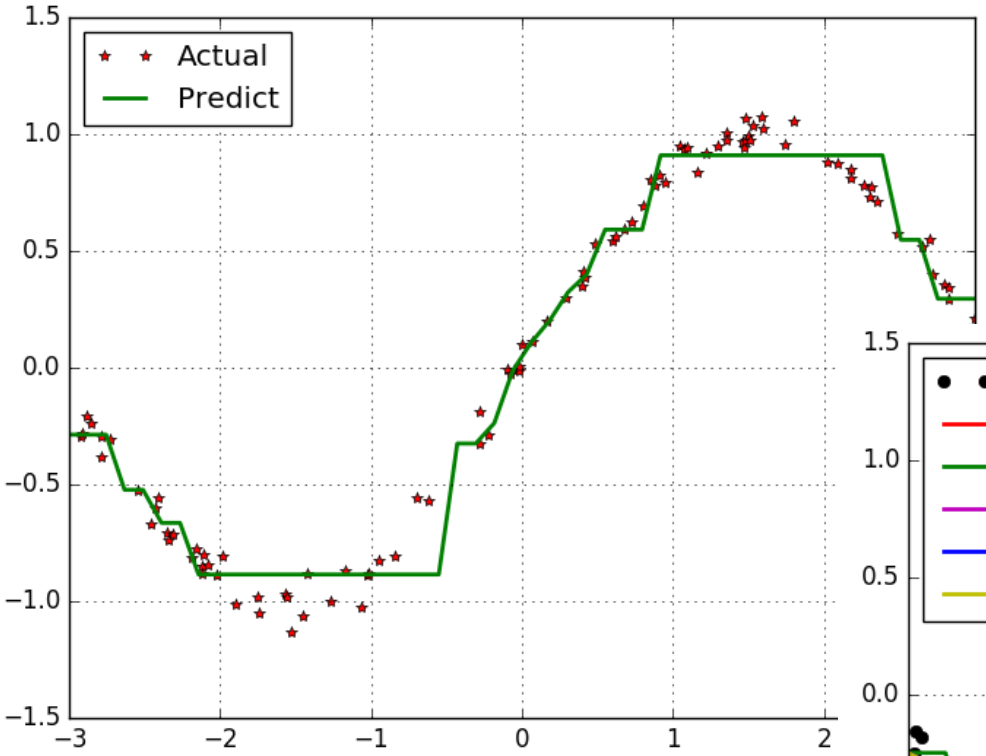


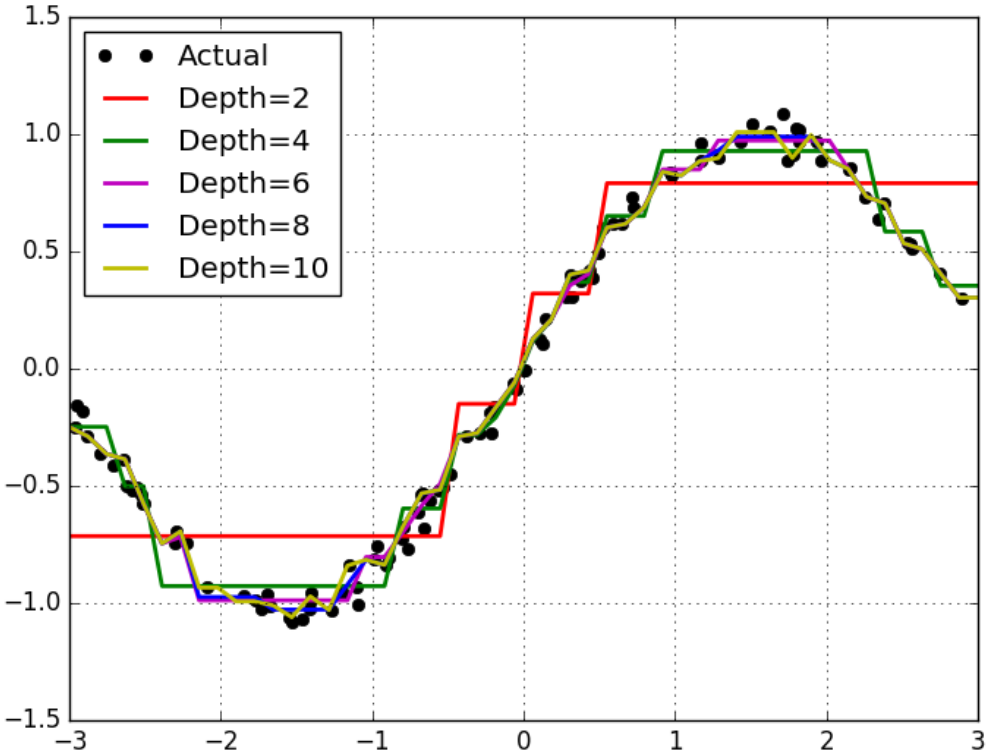
随机森林

- 50


鸢尾花数据二特征随机森林结果
花萼宽度
花萼长度
花瓣长度
花萼长度
花瓣宽度
花萼长度
花瓣长度
花萼宽度
花瓣宽度
花萼宽度
花瓣宽度
花瓣长度

决策树用于拟合


Actual
Predict
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−3
−2
−1
0
1
2


Actual
Depth=2
Depth=4
Depth=6
Depth=8
Depth=10
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−3
−2
−1
0
1
2
3

总结

- 树是最重要的数据结构，没有之一。
- 从根节点到叶子的过程，是逐渐细化精确的过程，所以，实践中往往作为海量数据索引。
- 强调：树只是数据结构，非存储结构。实践中，可以使用数组来存储树。
  - 思考实例Huffman编码、堆排序、并查集等。

我们在这里

- http://wenda.ChinaHadoop.cn
  - 视频/课程/社区
- 微博
  - @ChinaHadoop
  - @邹博_机器学习
- 微信公众号
  - 小象
  - 大数据分析挖掘

感谢大家！

恳请大家批评指正！

```
template<typename T>
class CTreeNode
{
public:
    T value;
    int bf;      //平衡因子
    int nData;   //额外数据
    CTreeNode<T>* lChild;
    CTreeNode<T>* rChild;

public:
    CTreeNode(const T& value);
    CTreeNode();
    int GetHeight() const;
    const T& GetLChild() const;
    const T& GetRChild() const;
};
```

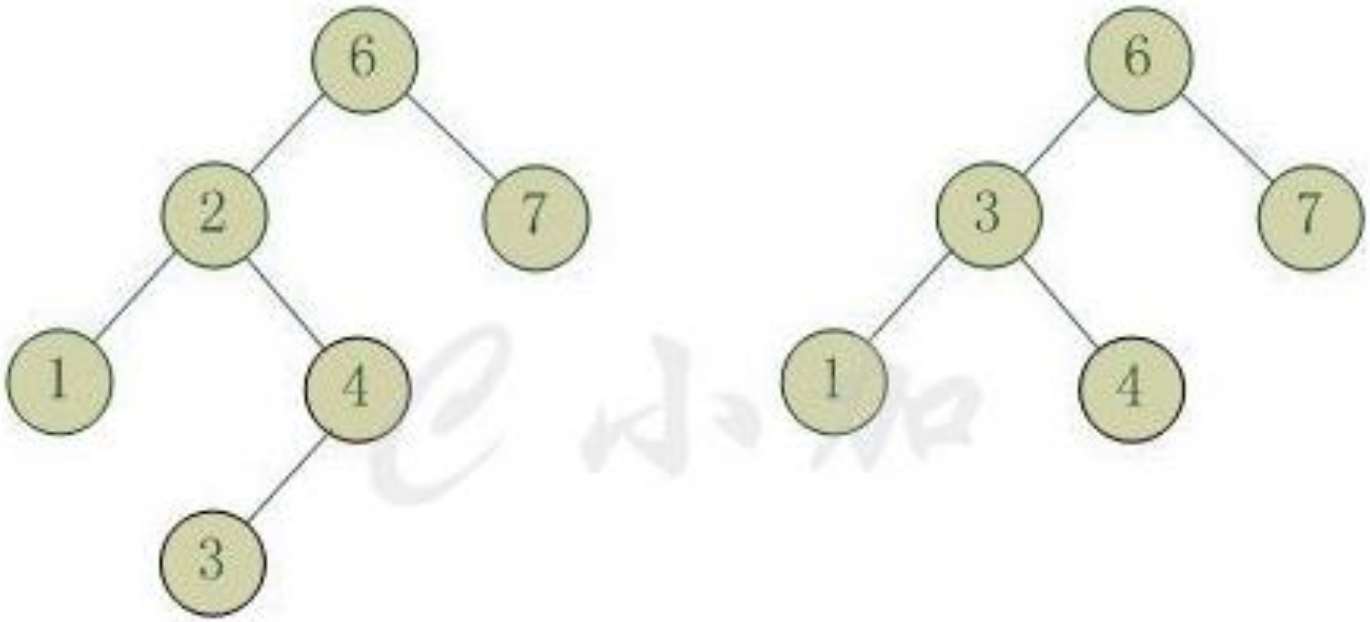
附：平衡二叉树

- 平衡二叉树是第一个引入平衡概念的二叉树
  - Balanced Binary Tree，1962年由G.M.Adelson Velsky、E.M.Landis发明，又称AVL树。
- 每个结点的左右子树的高度之差不超过1，
  - 如果插入或者删除结点后高差大于1，则进行结点旋转，重新维护平衡状态。
  - 解决了二叉查找树退化成链表的问题，插入、查找、删除的时间复杂度最坏情况是O(logN)。
    - 最好情况也是O(logN)

AVL实现举例

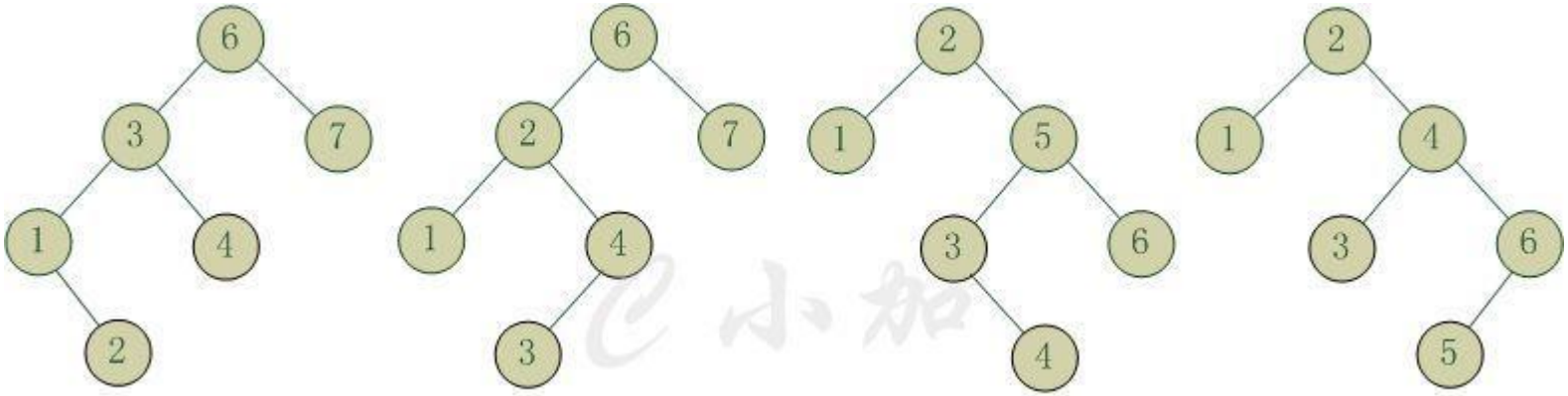
```
template<typename T>
class AFX_EXT_CLASS CBalanceTree
{
private:
    bool m_bBinaryTree;     //普通二叉树
    CTreeNode<T>* m_pRoot;
    CTreeNode<T>* m_pInsert;    //刚刚插入的接点
    int m_iNodeSize;    //结点数目
    bool m_bAllInsert;  //为true，则表示：如果插入时发现有值，则更新
    TCompare m_fCompare;

    bool m_bFastMode;
    CTreeNode<T>* m_pAllNode;
    int m_nNodeUsed;    //使用的结点

public:
    CBalanceTree();
    ~CBalanceTree();
    void SetAllInsert(bool bInsert);
    bool IsAllInsert() const;
    void SetBinaryTree(bool bBinary);   //设置为普通的二叉树
    bool IsBinaryTree() const;
    void Init();
    void Clear();   //删除整个树
    void SetCompare(TCompare funcCompare);
    bool Insert(const T& value);
    CTreeNode<T>* GetInsert();
    bool Delete(const T& value);
    bool Delete(IS_TEST funcIsTest, void* lpParam); //删除满足IS_TEST(lpParam)的结点
    bool GetValue(T& value) const;
    bool Search(const T& value) const;
    bool Search(const T& value, CTreeNode<T>*& pNode) const;
    const T& Min() const;
    void InOrder(FUNC func, void* lpParam) const;   //中序遍历
    void PreOrder(FUNC func, void* lpParam) const;  //前序遍历
    int GetNodeSize() const;    //获得树的结点数目
    bool IsEmpty() const;       //树的结点数目是否为0
    bool Swap(T t1, T t2);      //交换两个结点的值

    //快速模式
    void SetFastMode(bool bFast, int nInitNode);        //使用快速模式
    bool IsFastMode() const;
```

二叉查找树与平衡二叉树



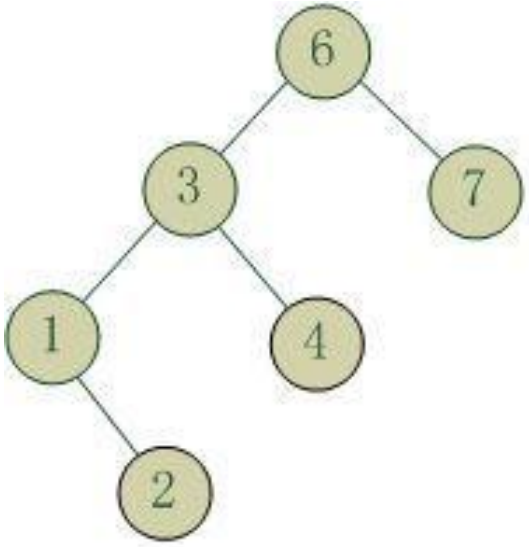
分析高度不平衡结点

- 维护不平衡的直接手段是选择不平衡结点X的某孩子C作为父结点，当前结点X作为原子结点C的子结点。这显然会影响到X的子结点、孙结点，因此，分成如下四种情况：
  - 高度不平衡结点的两颗子树的高度差2



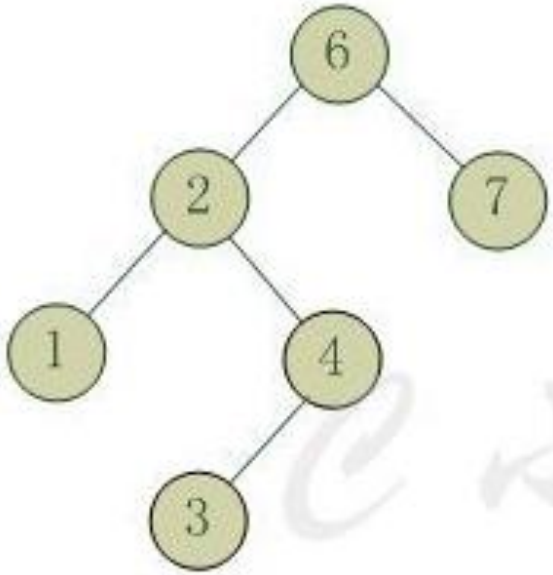
高度不平衡1：左左

- 6结点的左子树3结点高度比右子树7结点大2，左子树3结点的左子树1结点高度大于右子树4结点，这种情况称为左左。



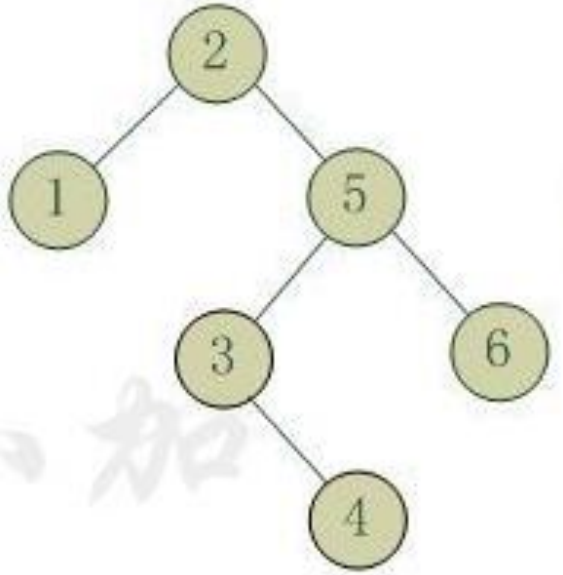
高度不平衡2：左右

- 6结点的左子树2结点高度比右子树7结点大2，左子树2结点的左子树1结点高度小于右子树4结点，这种情况成为左右。



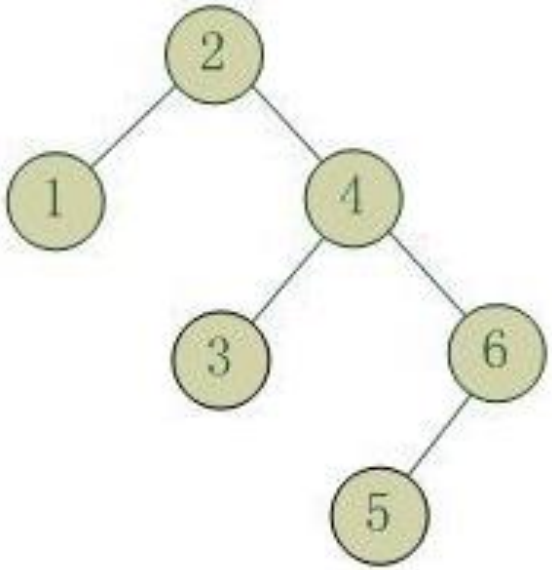
高度不平衡3：右左

- 2结点的左子树1结点高度比右子树5结点小2，右子树5结点的左子树3结点高度大于右子树6结点，这种情况成为右左。

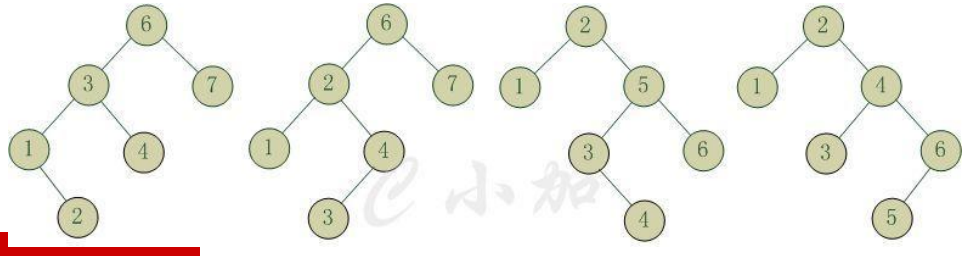


高度不平衡4：右右

- 2结点的左子树1结点高度比右子树4结点小2，右子树4结点的左子树3结点高度小于右子树6结点，这种情况成为右右。



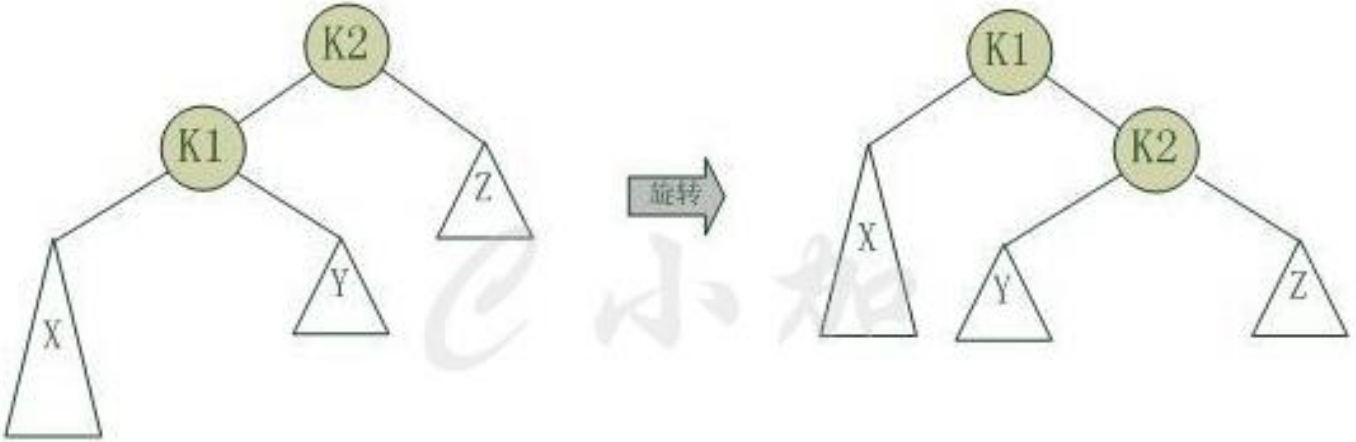
对称与旋转



- 左左和右右对称；左右和右左对称
- 左左和右右两种情况是对称的，这两种情况的旋转算法是一致的，只需要经过一次旋转就可以达到目标，称之为单旋转。
- 左右和右左两种情况也是对称的，这两种情况的旋转算法也是一致的，需要进行两次旋转，称之为双旋转。

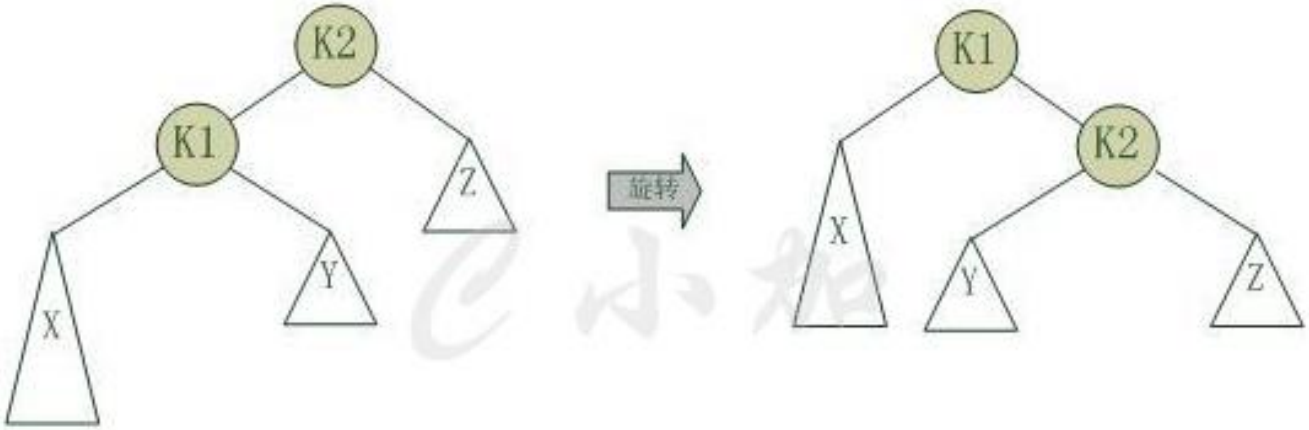
左左单旋转

- 结点K2不满足平衡特性，因为它的左子树k1比右子树Z深2层，而且K1子树中，更深的一层的是K1的左子树X子树，所以属于左左情况。



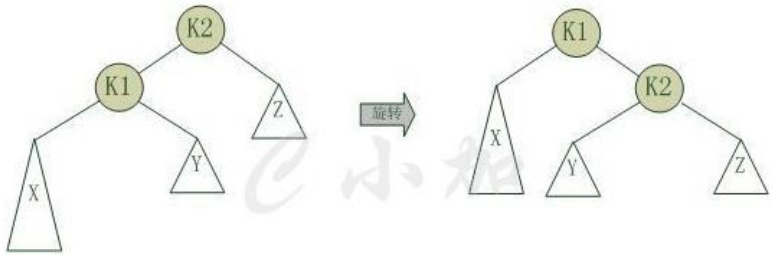
单旋转：对左孩子做旋转 (顺时针)

- 假设K2不平衡：中序遍历X-K1-Y-K2-Z
  - 把K1作为新的根结点
  - K2 > K1，所以，把K2置于K1的右子树上
  - K2 > Y > K1，所以，把Y置于K2的左子树上



AVL单旋转

- 把K1变成根结点
- 把K2置于K1的右子树
- 把Y置于K2的左子树

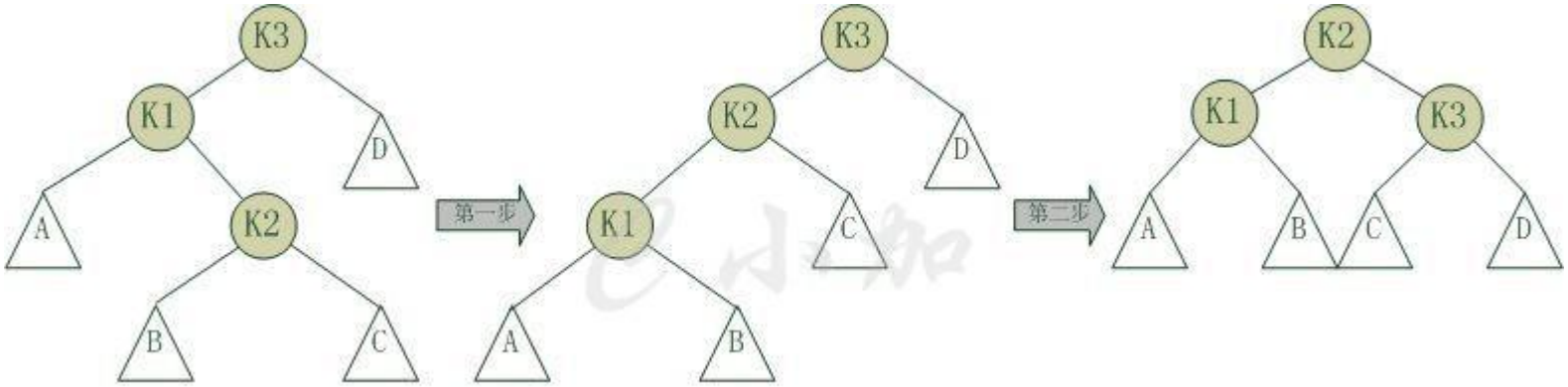



```
//左左情况下的旋转
template<class T>
void AVLTree<T>::SingRotateLeft(TreeNode<T>* &k2)
{
   TreeNode<T>* k1;
   k1=k2->lson;
   k2->lson=k1->rson;
   k1->rson=k2;
   k2->hgt=Max(height(k2->lson),height(k2->rson))+1;
   k1->hgt=Max(height(k1->lson),k2->hgt)+1;
   k2=k1;
}
```

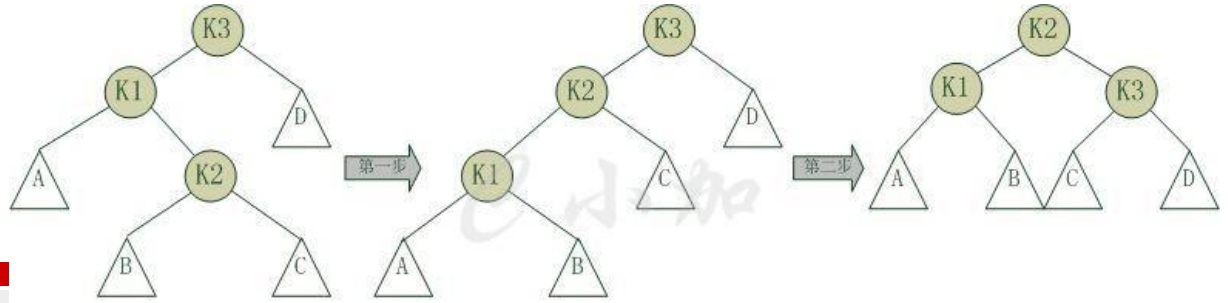
```
//右右情况下的旋转
template<class T>
void AVLTree<T>::SingRotateRight(TreeNode<T>* &k2)
{
   TreeNode<T>* k1;
   k1=k2->rson;
   k2->rson=k1->lson;
   k1->lson=k2;
   k2->hgt=Max(height(k2->lson),height(k2->rson))+1;
   k1->hgt=Max(height(k1->rson),k2->hgt)+1;
   k2=k1;
}
```

双旋转

- 对于左右和右左两种情况，单旋转不能使它达到一个平衡状态，要经过两次旋转。
- 左右：结点K3不平衡，左子树K1比右子树D深2层，且K1的子树中更深的是右子树K2。
  - K1右旋，然后K3左旋



Code




```
//左右情况的旋转
template<class T>
void AVLTree<T>::DoubleRotateLR(TreeNode<T>* &k3)
{
    SingRotateRight(k3->lson);
    SingRotateLeft(k3);
}
//右左情况的旋转
template<class T>
void AVLTree<T>::DoubleRotateRL(TreeNode<T>* &k3)
{
    SingRotateLeft(k3->rson);
    SingRotateRight(k3);
}
```

平衡二叉树的插入

- 插入的方法和二叉查找树基本一样，区别是，插入完成后需要从插入的结点开始维护一个到根结点的路径，每经过一个结点都要维持树的平衡。维持树的平衡要根据高度差的特点选择不同的旋转算法。

Code

```
template<class T>
void AVLTree<T>::insertpri(TreeNode<T>* &node,T x)
{
   if(node==NULL)//如果节点为空,就在此节点处加入x信息
   {
      node=new TreeNode<T>();
      node->data=x;
      return;
   }
   if(node->data>x)//如果x小于节点的值,就继续在节点的左子树中插入x
   {
      insertpri(node->lson,x);
      if(2==height(node->lson)-height(node->rson))
         if(x<node->lson->data)
            SingRotateLeft(node);
         else
            DoubleRotateLR(node);
   }
   else if(node->data<x)//如果x大于节点的值,就继续在节点的右子树中插入x
   {
      insertpri(node->rson,x);
      if(2==height(node->rson)-height(node->lson))//如果高度之差为2
         if(x>node->rson->data)
            SingRotateRight(node);
         else
            DoubleRotateRL(node);
   }
   else ++(node->freq);//如果相等,就把频率加1
   node->hgt=Max(height(node->lson),height(node->rson))+1 ;
}
//插入接口
template<class T>
void AVLTree<T>::insert(T x)
{
   insertpri(root,x);
}
```

平衡二叉树的查找

- 平衡二叉树使用和二叉查找树完全相同的查找方法，不过根据高度基本平衡存储的特性，平衡二叉树能保持O(logN)的稳定时间复杂度，而二叉查找树极端情况会退化成链表。

平衡二叉树的删除

- 删除的方法也和二叉查找树的一致，区别是，删除完成后，需要从删除结点的父亲开始向上维护树的平衡一直到根结点。

Code

```
template<class T>
void AVLTree<T>::Deletepri(TreeNode<T>* &node,T x)
{
   if(node==NULL) return ;//没有找到值是x的节点
   if(x < node->data)
   {
      Deletepri(node->lson,x);//如果x小于节点的值,就继续在节点的左子树中删除x
      if(2==height(node->rson)-height(node->lson))
        if(node->rson->lson!=NULL&&(height(node->rson->lson)>height(node->rson->rson)) )
           DoubleRotateRL(node);
        else
           SingRotateRight(node);
   }

   else if(x > node->data)
   {
      Deletepri(node->rson,x);//如果x大于节点的值,就继续在节点的右子树中删除x
      if(2==height(node->lson)-height(node->rson))
        if(node->lson->rson!=NULL&& (height(node->lson->rson)>height(node->lson->lson) ))
           DoubleRotateLR(node);
        else
           SingRotateLeft(node);
   }

   else//如果相等,此节点就是要删除的节点
   {
      if(node->lson&&node->rson)//此节点有两个儿子
      {
         TreeNode<T>* temp=node->rson;//temp指向节点的右儿子
         while(temp->lson!=NULL) temp=temp->lson;//找到右子树中值最小的节点
         //把右子树中最小节点的值赋值给本节点
         node->data=temp->data;
         node->freq=temp->freq;
         Deletepri(node->rson,temp->data);//删除右子树中最小值的节点
         if(2==height(node->lson)-height(node->rson))
         {
            if(node->lson->rson!=NULL&& (height(node->lson->rson)>height(node->lson->lson) ))
               DoubleRotateLR(node);
            else
               SingRotateLeft(node);
         }
      }
      else//此节点有1个或0个儿子
      {
         TreeNode<T>* temp=node;
         if(node->lson==NULL)//有右儿子或者没有儿子
         node=node->rson;
         else if(node->rson==NULL)//有左儿子
         node=node->lson;
         delete(temp);
         temp=NULL;
      }
   }
   if(node==NULL) return;
   node->hgt=Max(height(node->lson),height(node->rson))+1;
   return;
}
//删除接口
template<class T>
void AVLTree<T>::Delete(T x)
{
   Deletepri(root,x);
}
```

Code – part1

```
template<class T>
void AVLTree<T>::Deletepri(TreeNode<T>* &node,T x)
{
   if(node==NULL) return ;//没有找到值是x的节点
   if(x < node->data)
   {
       Deletepri(node->lson,x);//如果x小于节点的值,就继续在节点的左子树中删除x
       if(2==height(node->rson)-height(node->lson))
         if(node->rson->lson!=NULL&&(height(node->rson->lson)>height(node->rson->rson)) )
            DoubleRotateRL(node);
         else
            SingRotateRight(node);
   }

   else if(x > node->data)
   {
       Deletepri(node->rson,x);//如果x大于节点的值,就继续在节点的右子树中删除x
       if(2==height(node->lson)-height(node->rson))
         if(node->lson->rson!=NULL&& (height(node->lson->rson)>height(node->lson->lson) ))
            DoubleRotateLR(node);
         else
            SingRotateLeft(node);
   }
```

Code – part2

```
  else//如果相等,此节点就是要删除的节点
  {
     if(node->lson&&node->rson)//此节点有两个儿子
     {
        TreeNode<T>* temp=node->rson;//temp指向节点的右儿子
        while(temp->lson!=NULL) temp=temp->lson;//找到右子树中值最小的节点
        //把右子树中最小节点的值赋值给本节点
        node->data=temp->data;
        node->freq=temp->freq;
        Deletepri(node->rson,temp->data);//删除右子树中最小值的节点
        if(2==height(node->lson)-height(node->rson))
        {
           if(node->lson->rson!=NULL&& (height(node->lson->rson)>height(node->lson->lson) ))
              DoubleRotateLR(node);
           else
              SingRotateLeft(node);
        }
     }
     else//此节点有1个或0个儿子
     {
        TreeNode<T>* temp=node;
        if(node->lson==NULL)//有右儿子或者没有儿子
        node=node->rson;
        else if(node->rson==NULL)//有左儿子
        node=node->lson;
        delete(temp);
        temp=NULL;
     }
  }
  if(node==NULL) return;
  node->hgt=Max(height(node->lson),height(node->rson))+1;
  return;
}
//删除接口
template<class T>
void AVLTree<T>::Delete(T x)
{
  Deletepri(root,x);
}
```