# 1. 树

## 1.1 二叉排序树

二叉排序树(Binary Sort Tree),又称为二叉查找树。它或者是一颗空树，或者是具有下列性质的二叉树。

* 如果左子树不为空，那么左子树上所有节点的值均小于它的根的值。
* 如果右子树不为空，那么右子树上所有节点的值均大于它的根的值。
* 它的左子树、右子树也分别为二叉排序树。

从二叉树的定义可知，它的前提是二叉树，然后它采用了递归定义方法，再者，它的节点间满足一定的次序关系，左子树节点一定比双亲节点小，右子树的节点一定比双亲节点大。

构造一颗二叉排序树的目的，其实并不是为了排序，而是为了提高查找和插入、删除关键字的速度。不管怎么说，在一个有序数据集上的查找速度总是要快于无序的数据集，而二叉排序树这种非线性的结构也有利于插入和删除的实现。

### 1.1.1 二叉排序树初始化

二叉排序树数据结构:

|  |
| --- |
| //二叉排序树节点  template**<**class T**>**  class BinaryTreeNode**{**  public**:**  BinaryTreeNode**(**T **&**val**,** BinaryTreeNode **\***parent**);**  public**:**  T mData**;** //数据域  BinaryTreeNode **\***pParent**;** //父节点  BinaryTreeNode **\***pLchild**;** //左子树  BinaryTreeNode **\***pRchild**;** //右子树  **};**  //二叉排序树  template**<**class T**>**  class BinarySortTree**{**  public**:**  //构造  BinarySortTree**();**  //插入  void Insert**(**T val**);**  //删除  void Remove**(**T val**);**  //删除情况一 删除节点为叶子节点  void RemoveNoChildNode**(**BinaryTreeNode**<**T**>** **\***node**);**  //删除情况二 删除节点只有一个子节点  void RemoveSingleChildNode**(**BinaryTreeNode**<**T**>** **\***node**);**  //删除情况三 删除节点有两个子节点  void RemoveDoubleChildNode**(**BinaryTreeNode**<**T**>** **\***node**);**  //查找  BinaryTreeNode**<**T**>\*** Search**(**T val**);**  //获得最大值节点  BinaryTreeNode**<**T**>\*** GetMaxNode**(**BinaryTreeNode**<**T**>** **\***root**);**  //获得最小值节点  BinaryTreeNode**<**T**>\*** GetMinNode**(**BinaryTreeNode**<**T**>** **\***root**);**  //遍历  template**<**class \_Func**>** void Foreach**(**\_Func func**);**  template**<**class \_Func**>** void MyTraverse**(**BinaryTreeNode**<**T**>** **\***root**,** \_Func **&**func**);**  //释放  void MyRelease**(**BinaryTreeNode**<**T**>** **\***root**);**  **~**BinarySortTree**();**  public**:**  BinaryTreeNode**<**T**>** **\***pRoot**;** //二叉排序树根节点  int mSize**;** //二叉排序树节点数目  **};** |

二叉排序树初始化:

|  |
| --- |
| template**<**class T**>** BinarySortTree**<**T**>::**BinarySortTree**()**  **{**  **this->**pRoot **=** **NULL;**  **this->**mSize **=** 0**;**  **}** |

### 1.1.2 二叉排序树插入

|  |
| --- |
| //插入  template**<**class T**>** void BinarySortTree**<**T**>::**Insert**(**T val**)**  **{**  //节点已经存在，拒绝插入  **if** **(**Search**(**val**)){**  **return;**  **}**  //如果二叉排序树不为空时候  BinaryTreeNode**<**T**>** **\***pPrev **=** **NULL;**  BinaryTreeNode**<**T**>** **\***pCurrent **=** **this->**pRoot**;**  //查找插入节点的双亲节点  **while** **(**pCurrent **!=** **NULL){**  pPrev **=** pCurrent**;**  **if** **(**val **<** pCurrent**->**mData**){**  pCurrent **=** pCurrent**->**pLchild**;**  **}**  **else{**  pCurrent **=** pCurrent**->**pRchild**;**  **}**  **}**  //创建新节点  BinaryTreeNode**<**T**>** **\***node **=** **new** BinaryTreeNode**<**T**>(**val**,** pPrev**);**  //pPrev为空，表示当前二叉排序树为空树  **if** **(**pPrev **==** **NULL){**  **this->**pRoot **=** node**;**  **}**  **else{**  **if** **(**val **<** pPrev**->**mData**){**  pPrev**->**pLchild **=** node**;**  **}**  **else{**  pPrev**->**pRchild **=** node**;**  **}**  **}**  **this->**mSize**++;**  **}** |

### 1.1.3 二叉排序树查找

|  |
| --- |
| template**<**class T**>** BinaryTreeNode**<**T**>\*** BinarySortTree**<**T**>::**Search**(**T val**)**  **{**  BinaryTreeNode**<**T**>** **\***pNode **=** **this->**pRoot**;**  **while** **(**pNode **!=** **NULL){**  **if** **(!(**val **<** pNode**->**mData**)** **&&** **!(**pNode**->**mData **<** val**)){**  **return** pNode**;**  **}**  **else** **if** **(**val **<** pNode**->**mData**){**  pNode **=** pNode**->**pLchild**;**  **}**  **else{**  pNode **=** pNode**->**pRchild**;**  **}**  **}**  **return** **NULL;**  **}** |

### 1.1.4 二叉排序树删除

二叉树的查找和插入算法比较简单，但是对于二叉排序树的删除操作，就不那么容易了，由于我们不能因为删除一个节点，而让这棵树变得不满足二叉排序树的特性，所以删除需要考虑多种情况。

那么在删除二叉排序中的节点的时候，一般有三种情况需要我们考虑:

* 叶子节点;
* 仅有左子树或者右子树的节点;
* 左子树和右子树都存在;

这三种情况中，第三种较为复杂。虽然有这三种情况，但是我们还应该考虑下删除根节点和非根节点的情况，大体处理类似，但是细节还是不同。

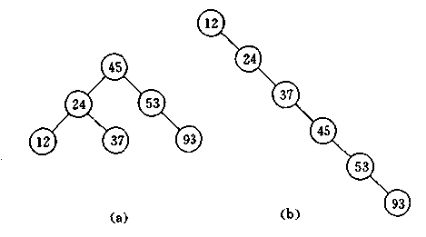
|  |
| --- |
| //删除情况一 删除节点为叶子节点  template**<**class T**>** void BinarySortTree**<**T**>::**RemoveNoChildNode**(**BinaryTreeNode**<**T**>** **\***node**)**  **{**  **if** **(NULL** **==** node**){**  **return;**  **}**  //如果删除的节点为根节点  **if** **(**node **==** **this->**pRoot**){**  **this->**pRoot **=** **NULL;**  **}**  **else{**  //删除节点不为根节点，则直接删除节点，并将此节点父节点设置为NULL  //删除节点父节点  BinaryTreeNode**<**T**>** **\***pParent **=** node**->**pParent**;**  **if** **(**pParent**->**pLchild **==** node**){**  pParent**->**pLchild **=** **NULL;**  **}**  **else{**  pParent**->**pRchild **=** **NULL;**  **}**  **}**  **}**  //删除情况二 删除节点只有一个子节点  template**<**class T**>** void BinarySortTree**<**T**>::**RemoveSingleChildNode**(**BinaryTreeNode**<**T**>** **\***node**)**  **{**  **if** **(NULL** **==** node**){**  **return;**  **}**  //如果删除节点为根节点  **if** **(this->**pRoot **==** node**){**  **this->**pRoot **=** node**->**pLchild **==** **NULL** **?** node**->**pRchild **:** node**->**pLchild**;**  **}**  **else{**  /\*如果删除节点不为根节点\*/  //删除节点父节点  BinaryTreeNode**<**T**>** **\***pParent **=** node**->**pParent**;**  //获得删除节点的子节点  BinaryTreeNode**<**T**>** **\***pChildNode **=** node**->**pLchild **==** **NULL** **?** node**->**pRchild **:** node**->**pLchild**;**  **if** **(**pParent**->**pLchild **==** node**){**  pParent**->**pLchild **=** pChildNode**;**  **}**  **else{**  pParent**->**pRchild **=** pChildNode**;**  **}**  **}**  **}**  //删除情况三 删除节点有两个子节点  template**<**class T**>** void BinarySortTree**<**T**>::**RemoveDoubleChildNode**(**BinaryTreeNode**<**T**>** **\***node**)**  **{**  **if** **(NULL** **==** node**){**  **return;**  **}**  //将删除节点的左子树最大值或者右子树最小值替换为当前节点  //获得删除节点的父节点  BinaryTreeNode**<**T**>** **\***pDelNodeParent **=** node**->**pParent**;**  //我们这里取左子树最大值  BinaryTreeNode**<**T**>** **\***pLeftMaxNode **=** **this->**GetMaxNode**(**node**->**pLchild**);**  //如果删除节点为根节点  **if** **(this->**pRoot **==** node**){**  //判断最大节点是否有左子树,如果有则将当前最大节点的右子树设置为最大节点的右子树  **if** **(**pLeftMaxNode**->**pLchild **!=** **NULL){**  pLeftMaxNode**->**pParent**->**pRchild **=** pLeftMaxNode**->**pLchild**;**  **}**  **else{**  pLeftMaxNode**->**pParent**->**pRchild **=** **NULL;**  **}**  //设置最大节点的左子树为根节点的左子树，最大节点的右子树为根节点的右子树  pLeftMaxNode**->**pLchild **=** node**->**pLchild**;**  pLeftMaxNode**->**pRchild **=** node**->**pRchild**;**  //设置当前根结点为最大值节点  **this->**pRoot **=** pLeftMaxNode**;**  **return;**  **}**    **if** **(**pLeftMaxNode **!=** node**->**pLchild**){**  //将左子树最大节点的父节点设置为NULL  pLeftMaxNode**->**pParent**->**pRchild **=** **NULL;**  //将左子树最大节点的左右子树设置为删除节点的左右子树  pLeftMaxNode**->**pLchild **=** node**->**pLchild**;**  pLeftMaxNode**->**pRchild **=** node**->**pRchild**;**  //将删除节点的父节点指向左子树最大节点  **if** **(**pDelNodeParent**->**pLchild **==** node**){**  pDelNodeParent**->**pLchild **=** pLeftMaxNode**;**  **}**  **else{**  pDelNodeParent**->**pRchild **=** pLeftMaxNode**;**  **}**  **}**  **else{**  //设置删除节点的父节点指向左子树最大节点  pLeftMaxNode**->**pRchild **=** node**->**pRchild**;**  //将删除节点的父节点指向左子树最大节点  **if** **(**pDelNodeParent**->**pLchild **==** node**){**  pDelNodeParent**->**pLchild **=** pLeftMaxNode**;**  **}**  **else{**  pDelNodeParent**->**pRchild **=** pLeftMaxNode**;**  **}**  **}**  **}**  //删除  template**<**class T**>** void BinarySortTree**<**T**>::**Remove**(**T val**)**  **{**  **if** **(this->**mSize **==** 0**){**  printf**(**"二叉排序树为NULL树!\n"**);**  **return;**  **}**  //查找删除节点  BinaryTreeNode**<**T**>** **\***pDelNode **=** Search**(**val**);**  **if** **(NULL** **==** pDelNode**){**  printf**(**"删除节点不存在!\n"**);**  **return;**  **}**  //情况一：删除的节点为叶子节点  **if** **(!**pDelNode**->**pLchild **&&** **!**pDelNode**->**pRchild**){**  RemoveNoChildNode**(**pDelNode**);**  **}**  //情况二：删除有两个孩子节点  **else** **if** **(**pDelNode**->**pLchild **&&** pDelNode**->**pRchild**){**  RemoveDoubleChildNode**(**pDelNode**);**  **}**  //情况三: 删除只有一个孩子节点  **else{**  RemoveSingleChildNode**(**pDelNode**);**  **}**  **this->**mSize**--;**  free**(**pDelNode**);**  pDelNode **=** **NULL;**  **}** |

### 1.1.5 二叉排序树销毁

|  |
| --- |
| template**<**class T**>** void BinarySortTree**<**T**>::**MyRelease**(**BinaryTreeNode**<**T**>** **\***root**)**  **{**  **if** **(NULL** **==** root**){**  **return;**  **}**  //释放当前节点的左子树  MyRelease**(**root**->**pLchild**);**  //释放当前节点的右子树  MyRelease**(**root**->**pRchild**);**  //释放当前节点  **delete** root**;**  **}**  template**<**class T**>** BinarySortTree**<**T**>::~**BinarySortTree**()**  **{**  MyRelease**(this->**pRoot**);**  **}** |

### 1.1.6 二叉排序树总结

二叉排序树是以链接的方式存储，保持了链式存储结构的插入和删除操作的时候不需要移动元素的有点，只要找到何时的插入位置后，仅需要修改指针的指向即可，插入和删除的时间性能比较好。而对于二叉派叙述的查找，走的就是从根节点到要查找的节点的路径，其比较次数等于给定值的节点在二叉排序树中的层数。极端情况下，最少为1次，即根节点就是要找的节点，最多也不会超过树的深度。也就是说，二叉排序树的查找性能取决于二叉排序树的形状。可问题就在于二叉排序树的形状是不确定的。



因此，如果我们希望对一个集合按二叉排序树查找，最好把它构建成一棵平衡的二叉排序树。这样就引出一个问题，如何让二叉派叙述平衡的问题。

## 1.2 平衡二叉树

平衡二叉树(Self-Balancing Binary Search Tree 或 Height-Balanced Binary Search Tree)，是一种二叉排序树，其中每一个节点的左子树和右子树高度差至多等于1.

我们将二叉树上节点的左子树高度减去右子树高度的值成为平衡因子BF(Balance Factor),那么平衡二叉树上所有节点的平衡因子只可能是-1、0、1.只要二叉树上有一个节点的平衡因子的绝对值大于1，则该二叉树就是不平衡的。

|  |
| --- |
| **平衡二叉树的前提是这棵树一定是一个二叉排序树。** |

## 1.3 AVL树

**AVL**是最先发明的自平衡二叉查找树算法。在AVL中任何节点的两个子树的高度最大差别为1，所以它也被称为高度平衡树。增加和删除可能需要通过一次或多次树旋转来重新平衡这个树。常用算法有红黑树、AVL、Treap、伸展树等。

我们在二叉排序树的基础上，新增了几个成员函数。

|  |
| --- |
| //更新二叉树各个节点深度  int UpdateTreeNodeDepth**(**BinaryTreeNode**<**T**>** **\***root**);**  //获得节点深度  int GetNodeDepth**(**BinaryTreeNode**<**T**>** **\***node**);**  //获得子树高度差  int GetNodeDepthDiff**(**BinaryTreeNode**<**T**>** **\***root**);**  //调整二叉排序树为平衡二叉树(参数为 新插入节点，从新插入节点回溯，寻找最小不平衡子树)  void AdjustBalancedBinaryTree**(**BinaryTreeNode**<**T**>** **\***node**);**  //左旋转  void LeftRotate**(**BinaryTreeNode**<**T**>** **\***node**);**  //右旋转  void RightRotate**(**BinaryTreeNode**<**T**>** **\***node**);** |

## 1.4 红黑树

红黑树（Red Black Tree） 是一种自平衡二叉查找树，是在[计算机](http://baike.baidu.com/view/3314.htm)科学中用到的一种[数据结构](http://baike.baidu.com/view/9900.htm)，典型的用途是实现[关联数组](http://baike.baidu.com/view/1654988.htm)。

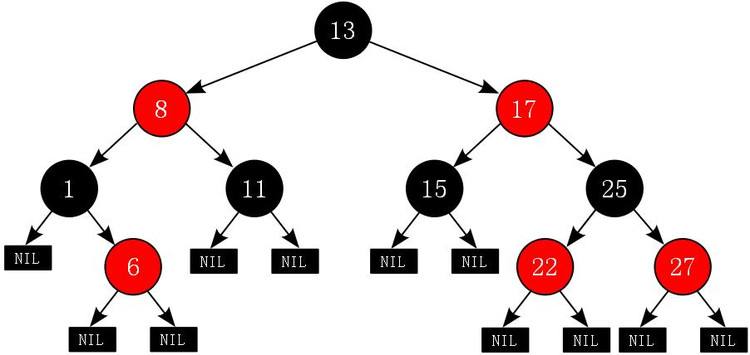
它是在1972年由Rudolf Bayer发明的，当时被称为平衡二叉B树（symmetric binary B-trees）。后来，在1978年被 Leo J. Guibas 和 Robert Sedgewick 修改为如今的“红黑树”。

红黑树和AVL树类似，都是在进行插入和删除操作时通过特定操作保持二叉查找树的平衡，从而获得较高的查找性能。

它虽然是复杂的，但它的最坏情况运行时间也是非常良好的，并且在实践中是高效的。

它的统计性能要好于[平衡二叉树](http://baike.baidu.com/view/593144.htm)(有些书籍根据作者姓名，Adelson-Velskii和Landis，将其称为AVL-树)，因此，红黑树在很多地方都有应用。在C++ STL中，很多部分(包括set, multiset, map, multimap)应用了红黑树的变体(SGI STL中的红黑树有一些变化，这些修改提供了更好的性能，以及对set操作的支持)。

|  |
| --- |
| 红黑树是每个节点都带有颜色属性的二叉查找树，颜色或红色或黑色。在二叉查找树强制一般要求以外，对于任何有效的红黑树我们增加了如下的额外要求:  性质1. 节点是红色或黑色。  性质2. 根节点是黑色。  性质3 每个叶节点（NIL节点，空节点）是黑色的。  性质4 每个红色节点的两个子节点都是黑色。(从每个叶子到根的所有路径上不能有两个连续的红色节点)  性质5. 从任一节点到其每个叶子的所有路径都包含相同数目的黑色节点。 |



当我们在对红黑树进行插入和删除等操作时，对树做了修改，那么可能会违背红黑树的性质。

为了保持红黑树的性质，我们可以通过对树进行旋转，即修改树中某些结点的颜色及指针结构，以达到对红黑树进行插入、删除结点等操作时，红黑树依然能保持它特有的性质（五点性质）。

# 2. 哈希表

## 2.1 概述

哈希表是种数据结构，它可以提供快速的插入操作和查找操作。它不以关键字的比较为基本操作，**采用直接寻址技术**。不论哈希表中有多少数据，插入和删除（有时包括侧除）**只需要接近常量的时间即0(1）的时间级**。实际上，这只需要几条机器指令。

对哈希表的使用者一一人来说，这是一瞬间的事。哈希表运算得非常快，在计算机程序中，如果需要在一秒种内查找上千条记录通常使用哈希表（例如拼写检查器)哈希表的速度明显比树快，树的操作通常需要O(N)的时间级。哈希表不仅速度快，编程实现也相对容易。

哈希表也有一些缺点它是基与数组的，数组创建后难于扩展某些哈希表被基本填满时，性能下降得非常严重，所以程序虽必须要清楚表中将要存储多少数据（或者准备好定期地把数据转移到更大的哈希表中，这是个费时的过程）。

而且，也没有一种简便的方法可以以任何一种顺序〔例如从小到大）遍历表中的数据项。如果需要这种能力，就只能选择其他数据结构。

然而如果不需要有序遍历数据，井且可以提前预测数据量的大小。那么哈希表在速度和易用性方面是无与伦比的。

散列函数能使对一个数据序列的访问过程更加迅速有效，通过散列函数，数据元素将被更快地定位：

## 2.2 哈希表的概念

设所有可能出现的关键字集合记为U(简称全集)。实际发生(即实际存储)的关键字集合记为K（|K|比|U|小得多）。

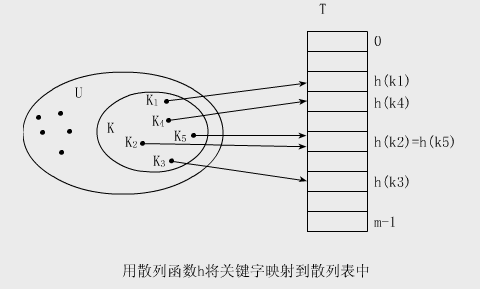
哈希方法是使用函数h将U映射到表T[0..m-1]的下标上（m=O(|U|)）。这样以U中关键字为自变量，以h为函数的运算结果就是相应结点的存储地址。从而达到在O(1)时间内就可完成查找。其中：

h：U→{0，1，2，…，m-1} ，通常称h为散列函数(Hash Function)。散列函数h的作用是压缩待处理的下标范围，使待处理的|U|个值减少到m个值，从而降低空间开销。

T为散列表(Hash Table)。

h(Ki)(Ki∈U)是关键字为Ki结点存储地址(亦称散列值或散列地址)。

将结点按其关键字的散列地址存储到散列表中的过程称为散列(Hashing)



哈希表的冲突现象

**冲突**

两个不同的关键字，由于散列函数值相同，因而被映射到同一表位置上。该现象称为冲突(Collision)或碰撞。发生冲突的两个关键字称为该散列函数的同义词(Synonym)。

【例】上图中的k2≠k5，但h(k2)=h(k5)，故k2和K5所在的结点的存储地址相同。

安全避免冲突的条件

最理想的解决冲突的方法是安全避免冲突。要做到这一点必须满足两个条件：

其一是|U|≤m

其二是选择合适的散列函数。

冲突不可能完全避免

通常情况下，h是一个压缩映像。虽然|K|≤m，但|U|>m，故无论怎样设计h，也不可能完全避免冲突。因此，只能在设计h时尽可能使冲突最少。同时还需要确定解决冲突的方法，使发生冲突的同义词能够存储到表中。

## 2.3 哈希函数构造方法

* 散列函数的选择有两条标准：
  + 简单

简单指散列函数的计算简单快速；

* + 均匀。

均匀指对于关键字集合中的任一关键字，散列函数能以等概率将其映射到表空间的任何一个位置上。也就是说，散列函数能将子集K随机均匀地分布在表的地址集{0，1，…，m-1}上，以使冲突最小化。

* 常用的哈希函数
* **余数法**

该方法是最为简单常用的一种方法。它是以表长m来除关键字，取其余数作为散列地址，即 **h(key)=key％m**

该方法的关键是选取m。选取的m应使得散列函数值尽可能与关键字的各位相关。m最好为素数。

【例】若选m是关键字的基数的幂次，则就等于是选择关键字的最后若干位数字作为地址，而与高位无关。于是高位不同而低位相同的关键字均互为同义词。

【例】若关键字是十进制整数，其基为10，则当m=100时，159，259，359，…，等均互为同义词。

* **直接地址法**

取关键字或关键字的的某个线性函数为哈希地址,即:

H(key) = key 或 H(key) = a \* key + b

其中a, b为常数(这种哈希函数叫做自身函数)

**例如:** 有一个从1岁到100岁的人口数字统计表,其中,年龄作为关键字,

哈希函数取关键字自身

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 地址 | 01 | 02 | … | 25 | … | 97 | 98 | 99 | 100 |
| 年龄 | 1 | 2 | … | 25 | … | 97 | 98 | 99 | 100 |
| 人数 | 3000 | 2000 | … | 5500 |  | … | … | … | … |

若要询问25岁的人有多少,只需要查表的第25项即可。

* **把关键字变为整数**

对于非整数的关键字key,如字符串 key = “abc”,可以将字符串中的每个字符在ASCII表中对应的整数值,然后相加,再用上面讲到的方法计算哈希地址即可。

key = “abc”

sum = 97 + 98 + 99 = 204

用余数法等方法对sum进行哈希地址计算。

## 2.4 处理冲突的方法

通常有两类方法处理冲突：开放定址(Open Addressing)法和链地址(Chaining)法。前者是将所有结点均存放在散列表T[0..m-1]中；后者通常是将互为同义词的结点链成一个单链表，而将此链表的头指针放在散列表T[0..m-1]中。

* 开放定址法

用开放定址法解决冲突的做法是：当冲突发生时，使用某种探查(亦称探测)技术在散列表中形成一个探查(测)序列。沿此序列逐个单元地查找，直到找到给定的关键字，或者碰到一个开放的地址(即该地址单元为空)为止（若要插入，在探查到开放的地址，则可将待插入的新结点存人该地址单元）。

Hi = (H(key) + di) % m i=1,2,3,4…k(k<=m-1)

其中H(key)为哈希函数,m为哈希表表长,di为增量序列

**如何获取增量序列?**

* **线性探测再散列**

**di = 1,2,3,4,5,…,m-1 (k≤m-1)**

* **二次探测再散列**

**di = 1²,-1²,2²,-2²,3²,…,±k²(k≤m/2)**

* **伪随机探测再散列**

**di = 伪随机数序列**

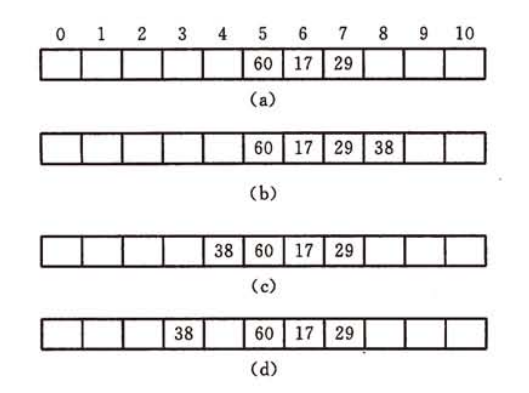
**例如:** 在长度11的哈希表中,已填有的关键字分别为17, 60, 29的记录(哈

希函数( H(key) = key % 11),现在有第四个记录,其关键字为38,计算起插

入到哈希表中的位置.如下图:

b: 线性探测再散列 c: 二次探测再散列

d: 伪随机探测再散列,伪随机数为9



* 再哈希法

在同义词产生地址冲突时计算另一个哈希函数地址，直到冲突不再发生。这种方法不易产生“聚集”，但增加了计算时间。

* 链地址法

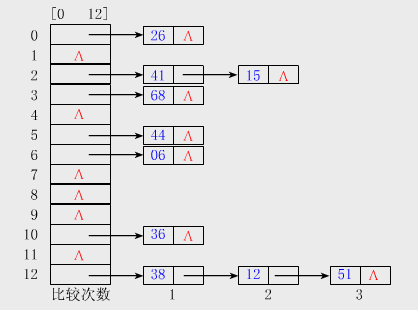
将所有关键字为同义词的结点链接在同一个单链表中。若选定的散列表长度为m，则可将散列表定义为一个由m个头指针组成的指针数组T[0..m-1]。凡是散列地址为i的结点，均插入到以T[i]为头指针的单链表中。T中各分量的初值均应为空指针。在拉链法中，装填因子α可以大于1，但一般均取α≤1。

【例9.2】已知一组关键字和选定的散列函数和3.5.2例图相同，用连地址法解决冲突构造这组关键字的散列表。

**解答：**取表长为13，故散列函数为h(key)=key％13，散列表为T[0..12]。

**注意：**

当把h(key)=i的关键字插入第i个单链表时，既可插入在链表的头上，也可以插在链表的尾上。这是因为必须确定key不在第i个链表时，才能将它插入表中，所以也就知道链尾结点的地址。若采用将新关键字插入链尾的方式，依次把给定的这组关键字插入表中，则所得到的散列表如下图所示。



# 3. 图

## 3.1 图的定义

线性表中，数据元素之间是1：1的关系，在树形结构中，数据元素之间是1：n多的关系，那么在图这种数据结构中，数据元素之间是n：n多的关系。

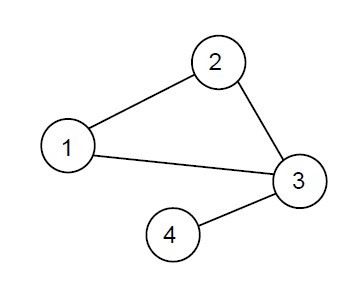
|  |
| --- |
| **图(Graph)是由顶点的有穷集合和顶点之间的关系(边)的集合组成，通常表示为：G(V,E),其中，G表示一个图，V是图中顶点的集合，E是图G中边的集合。** |

对于图的定义，我们需要明确几个注意的地方：

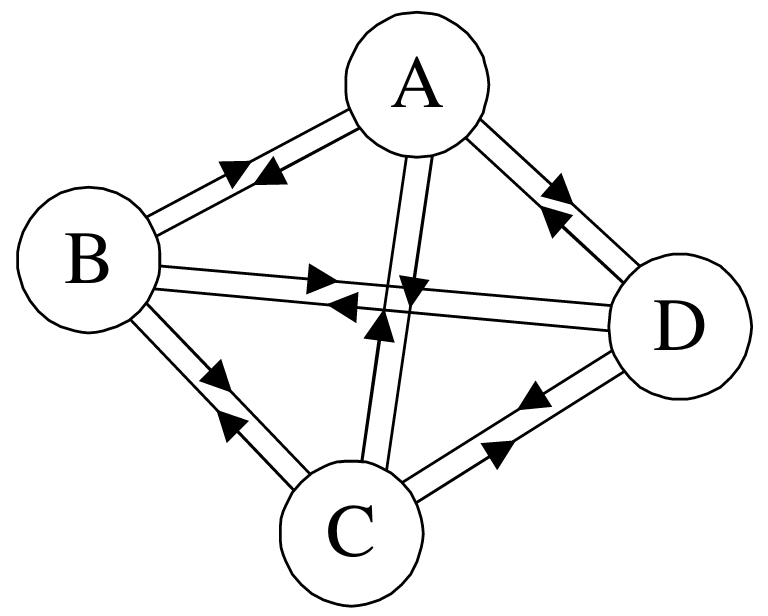
|  |
| --- |
| * 线性表中我们把数据称作元素，树中我们把数据元素叫做节点，在图中数据元素，我们称之为顶点(Vertex); * 线性表中可以没有数据元素，成为空表。树中没有节点，叫做空树。对于图，不允许没有顶点。在图的定义中，强调顶点集合是有穷非空，所以没有空图； * 线性表中，相邻的数据元素之间具有线性关系，树结构中相邻的两层节点具有层次关系，在图中，任意两个顶点之间都有可能有关系，顶点之间的逻辑关系用边来表示，边集可以是空的，但是顶点集合不能为空。 |

## 3.2 图的术语

如果顶点vi到顶点vj之间的边没有方向，则称这条边为无向边，用无序偶对(vi,vj)来表示。如果图中任意连个顶点之间的边都是无向边，则称该图为**无向图(Undirected Graph)**。



如果从顶点vi到顶点vj的边是有方向的，则称这条边为有向边，也称作弧(Arc)。用有序偶对<vi,vj>来表示，vi叫做弧尾(Tail)，vj叫做弧头(Head).如果图中任意两个顶点之间的边都是有向的，则称该图为**有向图(Directed Graph)**。



注意：无向图用小括号“()”表示，有向图用尖括号“<>”表示。

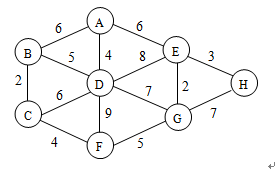
在图中，如果不存在顶点到自身的边，并且同一条边不重复出现，则称这样的图为**简单图**。我们主要讨论简单图。

在无向图中，如果任意两个顶点之间都存在边，则称该图为**无向完全图**。含有n个顶点的无向完全图有n(n-1)/2条边。

在有向图中，如果任意两个顶点之间都存在互为反方向的两条弧，则称该图为**有向完全图**。含有n个顶点的有向完全图有n(n-1)条边。

有很少条边或弧的图成为稀疏图，反之称为稠密图。这里的稀疏和稠密是个模糊的概念，都是相对而言。

有些图的边或弧具有与它相关的数字，这种与图的边或弧相关的数字叫做**权(weight).**这些权可以表示从一个顶点到另一个顶点的距离或者耗费。这种带权的图通常成为**网(Network)**



假设有两个图G=(v,{E}) 和 G1 = (V1,{E1})，如果V1属于V 并且E1属于E,则称G1是G的子图。

## 3.3 图顶点与边的关系

对于无向图G = (V,{E}),如果边(v,v1)属于E,则称顶点v和顶点v1互为邻接点(Adjacent),即v和v1相邻。顶点v的**度(Degree)**是和v相关联的边的数目，记为TD(v).

对于有向图G = (V,{E}),如果弧<v,v1>属于E，则称顶点v邻接到顶点v1，顶点v1顶点邻接自顶点v.以顶点v为头的弧的数目称为v的**入度(InDegree)**,记为ID(v);以v为尾的弧的数目成为v的**出度(OutDegree)**,记为OD(v);顶点v的度为TD(v) = ID(v) + OD(v).

无向图G = (V,{E})中从顶点v到顶点v1的路径(Path)是一个顶点序列.如果G是有向图，则路径也是有向的。

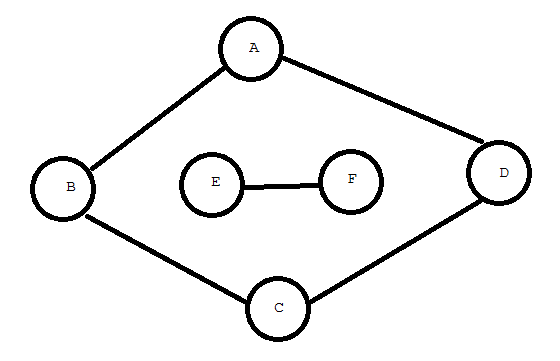
树中到任意节点的路径是唯一的，但是图中顶点与顶点之间的路径却不是唯一的。

**路径的长度**是路径上的边或弧的数目。

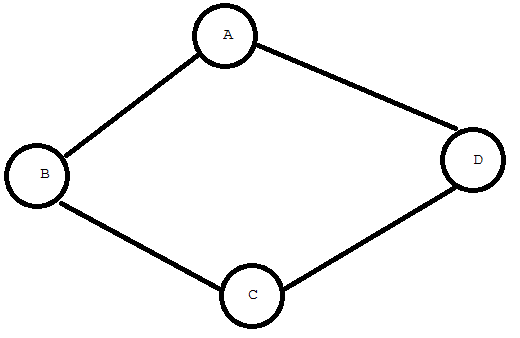
第一个顶点到最后一个顶点相同的路径称为回路或环(Cycle)。序列中顶点不重复出现的路径成为**简单路径**。除了第一个顶点和最后一个顶点之外，其余顶点不重复出现的回路，成为**简单回路或简单环**。

## 3.4 连通图相关术语

在无向图G中，如果从顶点v到顶点v1有路径，则称v和v1是连通的。如果对于图中任意两个顶点vi,vj都是连通的，则称G是**连通图**。



顶点A、B、C、D是连通的，但是顶点E、F没有路径，所以此图不是连通图。



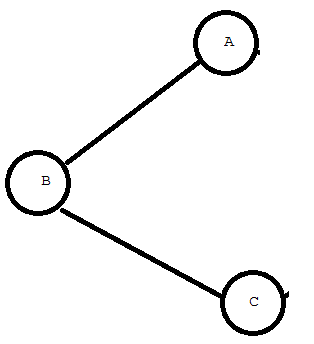
上图为连通图。

**无向图中的极大连通子图成为连通分量。注意连通分量的概念，它强调:**

|  |
| --- |
| * + 要是子图;   + 子图是连通的;   + 连通子图含有极大顶点数;   + 具有极大顶点数的连通子图包含依附于这些顶点的所有边; |



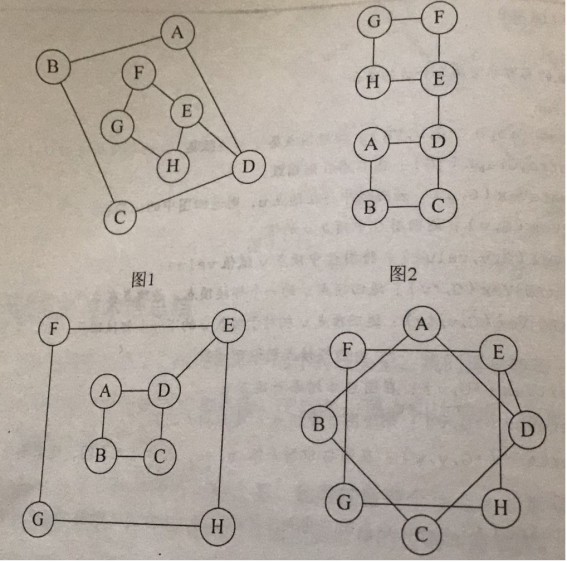
此图为连通分量。



此图虽然为子图，但是却不满足连通子图的极大顶点数，因此此图不是连通分量。

## 3.5 图的存储

图的存储结构比线性表和树的存储要更加复杂。首先，我门口头所说的“顶点的位置”或“临界点的位置”只是一个相对的概念。其实从图的逻辑结构定义来看，图上任何一个顶点都可被看称是第一个顶点，任一顶点邻接点之间不存在次序关系。



仔细发现，上图所画四个图其实是同一个图。也正由于图的结构比较复杂，任意两个顶点之间都可能存在联系，因此无法以数据元素在内存中的物理位置来表示元素之间的关系，也就是说不可能用简单的顺序存储结构来表示。而多重链表方式，以一个数据域和多个指针域组成的节点图中的一个顶点，虽然可以实现图结构，但是如果各个顶点的度数相差较大，按度数最大的顶点设计节点结构就会造成很多存储单元浪费，而按每个顶点自己的度数设计不同的顶点结构，又带来操作的不便。因此，对于图来说，如何对它实现物理存储是个难题，不过我们的前辈们已经解决了，现在我们就探讨下前辈门提供的几种不同的存储结构。

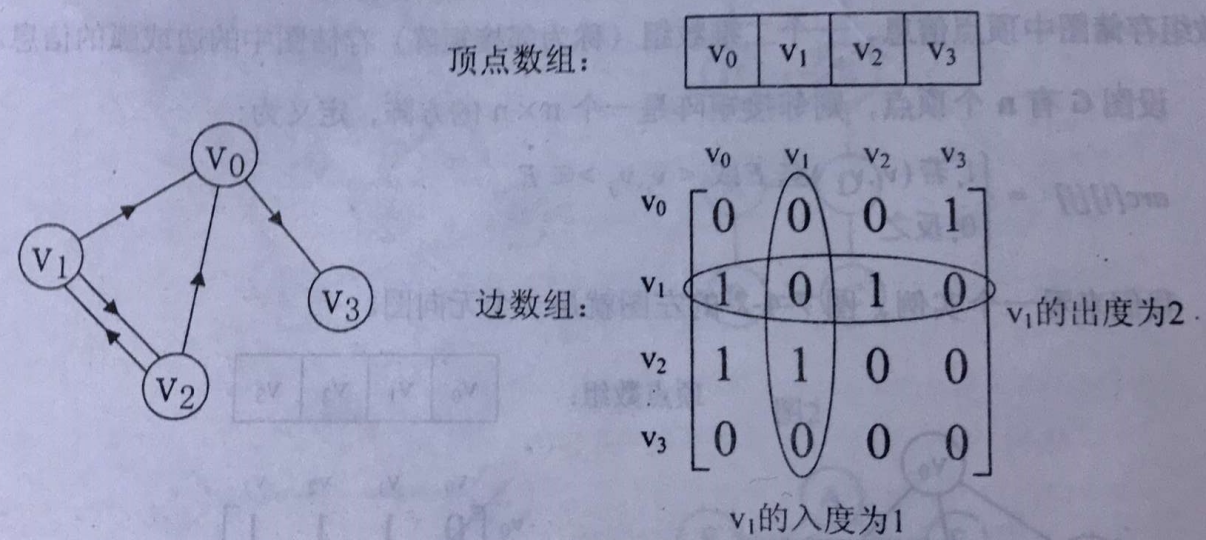
我们探讨其中两种存储方式,邻接矩阵和邻接表.

### 3.5.1 邻接矩阵

#### 3.5.1.1 邻接矩阵概念

考虑到图是由顶点和边或弧组成。合在一起比较困难，那就很自然地考虑到分两个结构来分别存储。顶点部分大小、主次，所以用一个一维数组来存储。而边或弧由于是顶点与顶点之间的关系，一维搞不定，那就考虑用一个二维数组来存储。

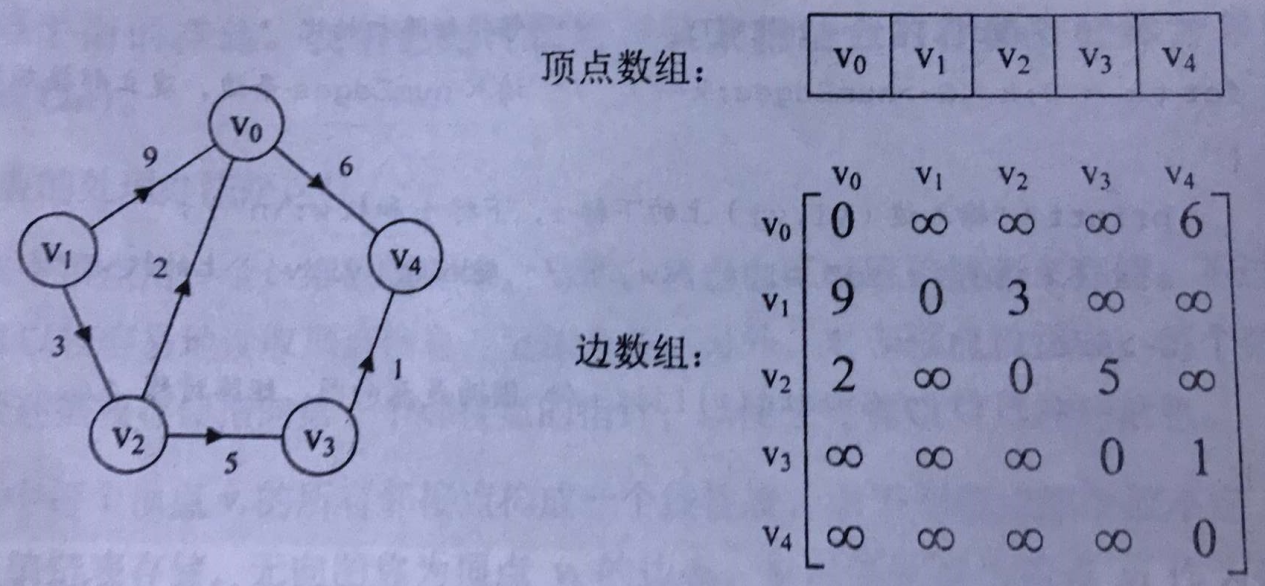
**图的邻接矩阵(Adjacency Matrix)存储方式是用两个数组来表示图。一个一维数组存储图中的顶点信息，一个二维数组(称作邻接矩阵)存储图中的边或弧的信息。**



有了这个矩阵，我们就很容易地直到地图中的信息：

|  |
| --- |
| * 我们要判断任意两个顶点是否有边无边非常容易; * 我们要直到某个顶点的度，其实就是这个顶点vi在邻接矩阵中第i行的元素之和; * 求顶点vi的所有邻接点就是将矩阵中第i行元素扫描一遍,arc[i][j]为1就是临界点; |

在图的术语中，我们提到了网的概念，也就是每条边上带有权的图叫做网。那么这些权值就要保存下来，如何处理这个需求呢？



#### 3.5.1.1 邻接矩阵实现

|  |
| --- |
| /\* numVex个定点 numEdges条边的无向网图 \*/  template**<**class VertexType**>**  class MyGraph**{**  public**:**  template**<**class MatrixType**>**  MyGraph**(**int numVex**,** int numEdges**,** VertexType **\***vexes**,** MatrixType edges**);**  public**:**  VertexType **\***pVertexes**;** //顶点数组  int **\*\***pEdges**;** //邻接矩阵，边集  int mNumVertexes**;** //顶点数  int mNumEdges**;** //边数  **};**  //初始化邻接矩阵  template**<**class VertexType**>** template**<**class MatrixType**>**  MyGraph**<**VertexType**>::**MyGraph**(**int numVex**,** int numEdges**,** VertexType **\***vexes**,** MatrixType edges**)**  **{**  **this->**mNumEdges **=** numEdges**;**  **this->**mNumVertexes **=** numVex**;**    //创建并初始化顶点数组  **this->**pVertexes **=** **new** VertexType**[**numVex**];**  **for** **(**int i **=** 0**;** i **<** numVex**;** **++**i**){**  **this->**pVertexes**[**i**]** **=** vexes**[**i**];**  **}**  //创建并初始化邻接矩阵  **this->**pEdges **=** **new** int**\*[this->**mNumVertexes**];**  **for** **(**int i **=** 0**;** i **<** **this->**mNumVertexes**;** **++**i**){**  **this->**pEdges**[**i**]** **=** **new** int**[this->**mNumVertexes**];**  **}**  **for** **(**int i **=** 0**;** i **<** **this->**mNumVertexes**;** **++**i**){**  **for** **(**int j **=** 0**;** j **<** **this->**mNumVertexes**;** **++**j**){**  **this->**pEdges**[**i**][**j**]** **=** edges**[**i**][**j**];**  **}**  **}**  **}** |

### 3.5.2 邻接表

#### 3.5.2.1 邻接表概念

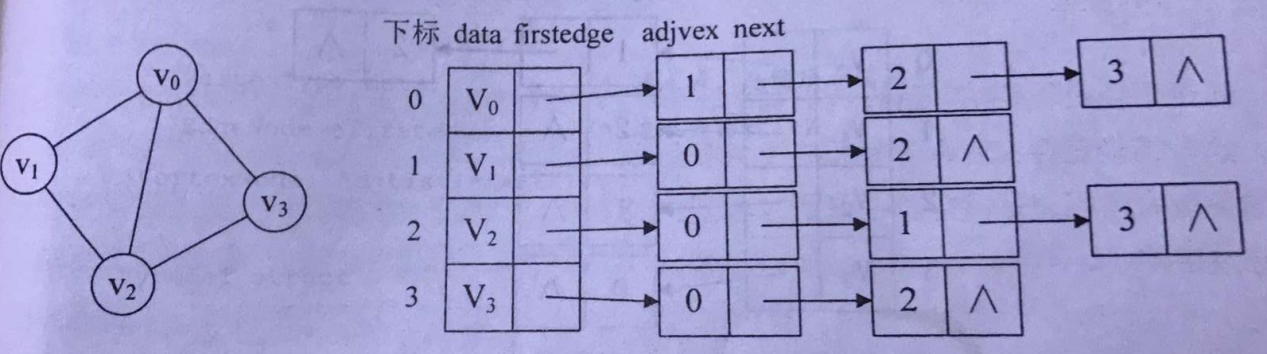
邻接矩阵是不错的一种图存储结构，但是我们发现对边数相对顶点较少的图，这种结构对存储空间极大浪费。

因此我们考虑另外一种存储结构方式。顺序存储结构就存在分配内存可能造成存储空间浪费的问题，于是引出了链式存储结构。同样的，我们可以考虑对边或者弧使用链式存储的方式来避免空间浪费问题。

我们使用数组来存储顶点，链表来存储边，我们把这种数组与链表相结合的存储方法称为邻接表(Adjacency List)。

邻接表的具体处理办法如下:

* + 图中顶点用一个一维数组存储，当然，顶点也可以使用单链表来存储，不过数组可以较容易地读取顶点信息，更加方便。另外，对于顶点数组中，每个数据元素还需要存储指向第一个邻接点的指针，以便于查找该顶点的边信息。
  + 图中每个顶点vi的所有邻接点构造一个线性表，由于邻接点的个数不定，所以用单链表存储，无向图称为顶点vi的边表，有向图则称为顶点vi做为弧尾的出边表。



#### 3.5.2.2 邻接表实现

|  |
| --- |
| //边表节点  class EdgeNode**{**  public**:**  int mAdjvex**;** //邻接点域  int mWeight**;** //权值  EdgeNode **\***pNext**;** //指向边表中下一个节点  **};**  //顶点节点  template**<**class VertexType**>**  class VertexNode**{**  public**:**  EdgeNode **\***pFirstEdge**;** //指向边表的第一个节点  VertexType mVertexData**;** //顶点数据  **};**  //邻接表  template**<**class VertexType**>**  class AdjacencyList**{**  public**:**  template**<**class MatrixType**>**  AdjacencyList**(**int numVex**,** int numEdges**,** VertexType **\***vexes**,** MatrixType edges**);**  public**:**  VertexNode**<**VertexType**>** **\***pVertexes**;** //指向顶点数组  int mNumVetexes**;** //顶点数  int mNumEdges**;** //边数  **};**  template**<**class VertexType**>** template**<**class MatrixType**>**  AdjacencyList**<**VertexType**>::**AdjacencyList**(**int numVex**,** int numEdges**,** VertexType **\***vexes**,** MatrixType edges**)**  **{**  **this->**mNumVetexes **=** numVex**;**  **this->**mNumEdges **=** numEdges**;**  //创建并初始化顶点数组  **this->**pVertexes **=** **new** VertexNode**<**VertexType**>[this->**mNumVetexes**];**  **for** **(**int i **=** 0**;** i **<** **this->**mNumVetexes**;** **++**i**){**  **this->**pVertexes**[**i**].**mVertexData **=** vexes**[**i**];**  **this->**pVertexes**[**i**].**pFirstEdge **=** **NULL;**  **}**  //初始化边集  **for** **(**int i **=** 0**;** i **<** **this->**mNumVetexes**;** **++**i**){**  **for** **(**int j **=** 0**;** j **<** **this->**mNumVetexes**;** **++**j**){**    //说明顶点i 和顶点j之间有边  **if** **(**edges**[**i**][**j**]** **>** 0 **&&** edges**[**i**][**j**]** **<** INT\_MAX**){**    //创建新边节点  EdgeNode **\***pNode **=** **new** EdgeNode**;**  pNode**->**mAdjvex **=** j**;** //邻接点  pNode**->**mWeight **=** edges**[**i**][**j**];** //权值  pNode**->**pNext **=** **this->**pVertexes**[**i**].**pFirstEdge**;**  **this->**pVertexes**[**i**].**pFirstEdge **=** pNode**;**  **}**  **}**  **}**  **}** |

## 3.6 图的遍历

图的遍历和树的遍历类似，从图中某一顶点出发访问遍图中其余顶点，且使每一个顶点仅被访问一次，这一过程叫做图的遍历(Traversing Graph)。

由于图中的任何顶点都可能和其余所有的顶点相邻接，极有可能沿着某条路径搜索后，又回到原顶点，而有些顶点可能还没有遍历到。因此我们需要在遍历图的过程中把访问过的顶点打上标记，以免多次访问同一个顶点。具体办法是设置一个访问数组visited[n]，n是图中顶点的个数。

对于图的遍历，如何避免因回路陷入死循环，就需要科学地设计遍历方案，通常有两种遍历次序方案：**深度优先遍历**和**广度优先遍历**。

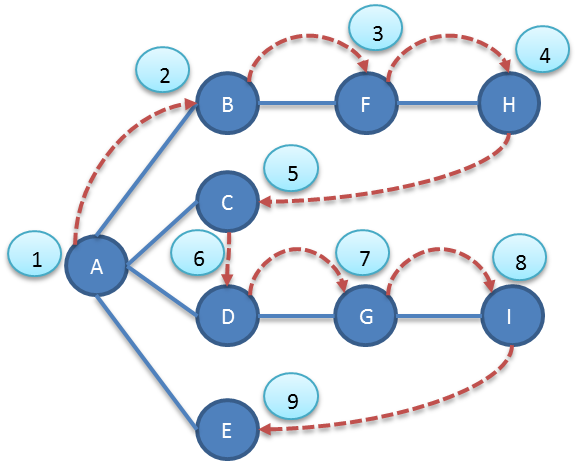
### 3.6.1 深度优先遍历

深度优先遍历(Depth First Search),也可以称为深度优先搜索，简称DFS.

它从图中某个结点v出发，访问此顶点，然后从v的未被访问的邻接点出发深度优先遍历图，直至图中所有和v有路径相通的顶点都被访问到。若图中尚有顶点未被访问，则另选图中一个未曾被访问的顶点作起始点，重复上述过程，直至图中的所有顶点都被访问到为止。

深度优先搜索是通过栈来实现的。

下图中的数字显示了深度优先搜索顶点被访问的顺序



为了实现深度优先搜索，首先选择一个起始顶点并需要遵守三个规则：

* 如果可能，访问一个邻接的未访问顶点，标记它，并把它放入栈中。
* 当不能执行规则1时，如果栈不空，就从栈中弹出一个顶点。
* 如果不能执行规则1和规则2，就完成了整个搜索过程。

邻接矩阵深度优先遍历:

|  |
| --- |
| template**<**class VertexType**>** void MyGraph**<**VertexType**>::**DFS**(**int i**,**bool **\***visited**)**  **{**  visited**[**i**]** **=** **true;**  cout **<<** **this->**pVertexes**[**i**]** **<<** " "**;**  **for** **(**int j **=** 0**;** j **<** **this->**mNumVertexes**;** **++**j**){**  **if** **(!**visited**[**j**]** **&&** **(**pEdges**[**i**][**j**]** **>** 0 **&&** pEdges**[**i**][**j**]** **<** INT\_MAX**)){**  DFS**(**j**,**visited**);**  **}**  **}**  **}**  template**<**class VertexType**>** void MyGraph**<**VertexType**>::**DepthFirstSearch**()**  **{**  //创建并且初始化visited数组  bool **\***visited **=** **new** bool**[this->**mNumVertexes**];**  **for** **(**int i **=** 0**;** i **<** **this->**mNumVertexes**;** **++**i**){**  visited**[**i**]** **=** **false;** //所有节点设置未被访问  **}**  **for** **(**int i **=** 0**;** i **<** **this->**mNumVertexes**;** **++**i**){**  //如果当前节点没有访问,则访问  **if** **(!**visited**[**i**]){**  DFS**(**i**,** visited**);**  **}**  **}**  **}** |

邻接表深度优先遍历:

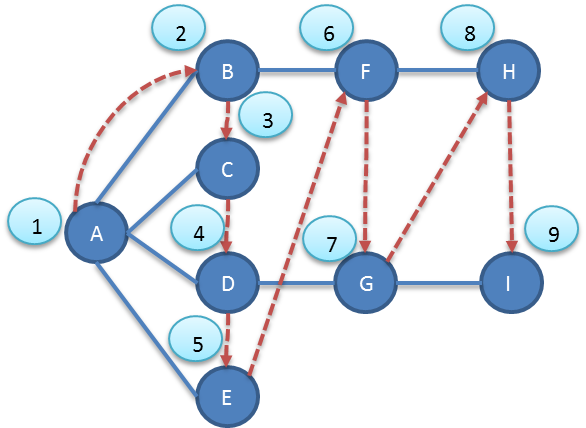
|  |
| --- |
| //深度优先遍历  template**<**class VertexType**>** void AdjacencyList**<**VertexType**>::**DFS**(**int i**,** bool **\***visited**)**  **{**  visited**[**i**]** **=** **true;**  cout **<<** **this->**pVertexes**[**i**].**mVertexData **<<** " "**;**    EdgeNode **\***pEdgeNode **=** **this->**pVertexes**[**i**].**pFirstEdge**;**  **while** **(**pEdgeNode **!=** **NULL){**  **if** **(!**visited**[**pEdgeNode**->**mAdjvex**]){**  DFS**(**pEdgeNode**->**mAdjvex**,** visited**);**  **}**  pEdgeNode **=** pEdgeNode**->**pNext**;**  **}**  **}**  template**<**class VertexType**>** void AdjacencyList**<**VertexType**>::**DepthFirstSearch**()**  **{**  bool **\***visited **=** **new** bool**[this->**mNumVetexes**];**  **for** **(**int i **=** 0**;** i **<** **this->**mNumVetexes**;** **++**i**){**  visited**[**i**]** **=** **false;**  **}**  **for** **(**int i **=** 0**;** i **<** **this->**mNumVetexes**;** **++**i**){**  //如果当前节点没有访问,则访问  **if** **(!**visited**[**i**]){**  DFS**(**i**,** visited**);**  **}**  **}**  **if** **(**visited **!=** **NULL){**  **delete[]** visited**;**  visited **=** **NULL;**  **}**  **}** |

深度优先遍历其实就是一个递归的过程。它从图中的某个顶点v出发，访问此顶点，然后从v的未被访问过的邻接点触发深度优先遍历图，直至图中所有和v有路径相同的顶点都被访问到。事实上，我们这里讲的是连通图，对于非连通图，只需要对它的连通分量分别进行深度优先遍历，即在先前一个顶点进行一次深度优先遍历后，如果图中尚有顶点未被访问，则另选图中一个未曾被访问的顶点作起始点，重复上述过程，直至图中所有的顶点都被访问到为止。

### 3.6.2 广度优先遍历

广度优先遍历(Breadth First Search),又称广度优先搜索，简称BFS.对图的广度优先遍历类似与对树的层序遍历。 在深度优先搜索中，算法表现得好像要尽快地远离起始点似的。相反，在广度优先搜索中，算法好像要尽可能地靠近起始点。它首先访问起始顶点的所有邻接点，然后再访问较远的区域。它是用队列来实现的。

下面图中的数字显示了广度优先搜索顶点被访问的顺序。



实现广度优先搜索，也要遵守三个规则：

* 访问下一个未来访问的邻接点，这个顶点必须是当前顶点的邻接点，标记它，并把它插入到队列中。
* 如果因为已经没有未访问顶点而不能执行规则1时，那么从队列头取一个顶点，并使其成为当前顶点。
* 如果因为队列为空而不能执行规则2，则搜索结束。

邻接矩阵广度优先遍历:

|  |
| --- |
| //广度优先遍历  template**<**class VertexType**>** void MyGraph**<**VertexType**>::**BFSTraverse**()**  **{**  //初始化队列  std**::**queue**<**VertexType**>** myqueue**;**  //visited数组  bool **\***visited **=** **new** bool**[this->**mNumVertexes**];**  **for** **(**int i **=** 0**;** i **<** **this->**mNumVertexes**;** **++**i**){**  visited**[**i**]** **=** **false;**  **}**  //对每一个顶点做广度优先遍历  **for** **(**int i **=** 0**;** i **<** **this->**mNumVertexes**;** **++**i**){**  //如果当前节点未被访问  **if** **(!**visited**[**i**]){**  //访问节点  visited**[**i**]** **=** **true;**  cout **<<** **this->**pVertexes**[**i**]** **<<** " "**;**  //将当前节点入队列  myqueue**.**push**(**i**);**  //将所有与i邻接点加入队列  **while** **(!**myqueue**.**empty**()){**  int vexID **=** myqueue**.**front**();**  myqueue**.**pop**();**  //遍历当前顶点邻接顶点  **for** **(**int j **=** 0**;** j **<** **this->**mNumVertexes**;** **++**j**){**  **if** **(!**visited**[**j**]** **&&**  **(**pEdges**[**vexID**][**j**]** **>** 0 **&&** pEdges**[**vexID**][**j**]** **<** INT\_MAX**)){**  //访问当前顶点  visited**[**j**]** **=** **true;**  cout **<<** **this->**pVertexes**[**j**]** **<<** " "**;**  //并将此顶点加入队列，方便下次获得此顶点互为邻接点的顶点  myqueue**.**push**(**j**);**  **}**  **}**  **}**  **}**  **}**  **if** **(**visited **!=** **NULL){**  **delete[]** visited**;**  visited **=** **NULL;**  **}**  **}** |

邻接表广度优先遍历：

|  |
| --- |
| //广度优先遍历  template**<**class VertexType**>** void AdjacencyList**<**VertexType**>::**BFSTraverse**()**  **{**  //辅助队列  std**::**queue**<**int**>** myqueue**;**  //创建初始化标记数组  bool **\***visited **=** **new** bool**[this->**mNumVetexes**];**  **for** **(**int i **=** 0**;** i **<** **this->**mNumVetexes**;** **++**i**){**  visited**[**i**]** **=** **false;**  **}**  //对每一个顶点进行广度优先遍历  **for** **(**int i **=** 0**;** i **<** **this->**mNumVetexes**;** **++**i**){**    **if** **(!**visited**[**i**]){**    //访问当前节点  visited**[**i**]** **=** **true;**  cout **<<** **this->**pVertexes**[**i**].**mVertexData **<<** " "**;**  //将当前节点加入到队列中  myqueue**.**push**(**i**);**  **while** **(!**myqueue**.**empty**()){**    int vexID **=** myqueue**.**front**();**  myqueue**.**pop**();**  EdgeNode **\***pNode **=** **this->**pVertexes**[**vexID**].**pFirstEdge**;**  **while** **(**pNode **!=** **NULL){**    **if** **(!**visited**[**pNode**->**mAdjvex**]){**    //访问当前节点  visited**[**pNode**->**mAdjvex**]** **=** **true;**  cout **<<** pVertexes**[**pNode**->**mAdjvex**].**mVertexData **<<** " "**;**    **}**  pNode **=** pNode**->**pNext**;**  **}**    **}**  **}**    **}**  **if** **(**visited **!=** **NULL){**  **delete[]** visited**;**  visited **=** **NULL;**  **}**  **}** |

对比图的深度优先遍历算法和广度优先遍历算法，两者的时间复杂度是一样的，不同之处在于对顶点的访问的顺序不同。可两者在全图遍历上没优劣之分，只是视情况不同选择不同算法。

如果图的顶点和边非常多，不能在段时间内遍历完成，遍历的目的是为了寻找何时的顶点，那么选择那种遍历要考虑以下。深度优先更适合目标比较明确的，以找到目标为主要目的的情况，而广度优先遍历更适合在不断扩大遍历范围时找到相对最优解的情况。

## 最小生成树

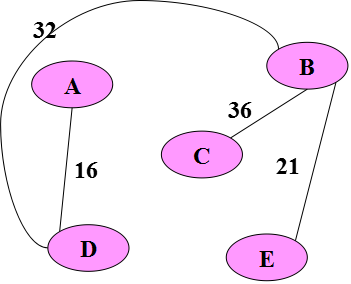
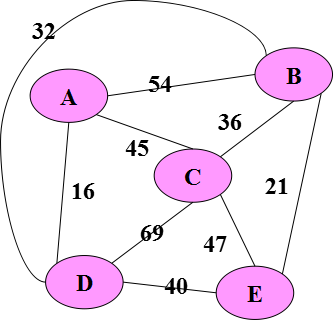
### 3.7.1 相关概念

**生成树**：一个**连通图的生成树是它的极小连通子图**，在**n个顶点的情形下，有n-1条边**。生成树是对连通图而言的，是连同图的极小连通子图，包含图中的所有顶点，有且仅有n-1条边。

**最小生成树：**在图论中，常常将树定义为一个无回路连通图。对于一个带权的无向连通图，其每个生成树所有边上的权值之和可能不同，我们把**所有边上权值之和最小的生成树称为图的最小生成树**。

**案例（造价最优问题就是一个最小生成树问题）**

|  |
| --- |
| 铺设煤气管道问题（图形结构）假设要在某个城市的n个居民区之间铺设煤气管道，则在这n个居民区之间只要铺设n-1条管道即可。假设任意两个居民区之间都可以架设管道，管道铺设方案。在众多可选边中，如何选择n-1条边，使总代价最小？这就是求该网络最小生成树问题。 |



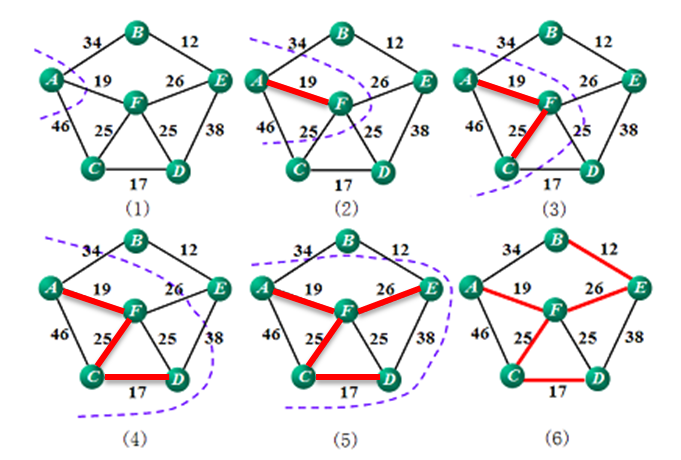
### 3.7.2 普里姆(Prim)算法

* + 基本思想：设G=(V, E)是具有n个顶点的连通网，T=(U, TE)是G的最小生成树，T的初始状态为U={u0}（u0∈V），TE={}，重复执行下述操作：**在所有u∈U，v∈V-U的边中找一条代价最小的边(u, v)并入集合TE，同时v并入U，直至U=V**。即：
    - 从连通网络 G = { V, E }中的某一顶点 u0 出发，选择与它关联的具有最小权值的边(u0, v)，将其顶点加入到生成树的顶点集合U中。
    - 以后每一步从**一个顶点在U中，而另一个顶点不在U中的各条边中选择权值最小的边(**u, v),把它的顶点加入到集合U中。如此继续下去，直到网络中的所有顶点都加入到生成树顶点集合U中为止。

此时，**TE中必有n-1条边，T=(V，TE)为G的最小生成树。**

**Prim算法的核心:始终保持TE中的边集构成一棵生成树。**

* + 示例



## 图的最短路径

### 3.8.1 基本概念

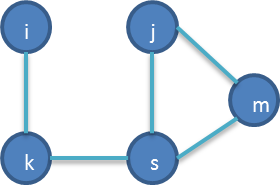
**最短路径**：从图中某一顶点（源点）到达另一顶点（终点）找到一条路径，沿此路径上各边的权值总和（称为路径长度）达到最小。   
**单源最短路径**：已知有向带权图(简称有向网)G=(V，E)，找出从某个源点s∈V到V中其余各顶点的最短路径。

习惯上称路径开始顶点为源点，路径的最后一个顶点为终点。

最短路径的最优子结构性质

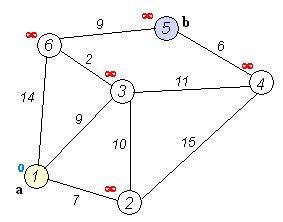
该性质描述为：如果P(i,j)={Vi....Vk..Vs...Vj}是从顶点i到j的最短路径，k和s是这条路径上的一个中间顶点，那么P(k,s)必定是从k到s的最短路径。下面证明该性质的正确性。

假设P(i,j)={Vi....Vk..Vs...Vj}是从顶点i到j的最短路径，则有P(i,j)=P(i,k)+P(k,s)+P(s,j)。而P(k,s)不是从k到s的最短距离，那么必定存在另一条从k到s的最短路径P'(k,s)，那么P'(i,j)=P(i,k)+P'(k,s)+P(s,j)<P(i,j)。则与P(i,j)是从i到j的最短路径相矛盾。因此该性质得证。(如图)



### 3.8.2 Dijkstra算法

迪杰斯特拉算法是由荷兰计算机科学家[狄克斯特拉](http://baike.baidu.com/view/156673.htm)于1959 年提出的，因此又叫[狄克斯特拉算法](http://baike.baidu.com/view/2541415.htm)。是从一个顶点到其余各顶点的[最短路径](http://baike.baidu.com/view/349189.htm)算法，解决的是有向图中最短路径问题。迪杰斯特拉算法主要特点是以起始点为中心向外层层扩展，直到扩展到终点为止。



**注意：它不能用来解决存在负权边的图。**

* 迪杰斯特拉(Dijkstra)算法

由上述性质可知，如果存在一条从i到j的最短路径(Vi.....Vk,Vj)，Vk是Vj前面的一顶点。那么(Vi...Vk)也必定是从i到k的最短路径。为了求出最短路径，Dijkstra就提出了**以最短路径长度递增，逐次生成最短路径的算法**。譬如对于源顶点V0，首先选择其直接相邻的顶点中长度最短的顶点Vi，那么当前已知可得从V0到达Vj顶点的最短距离dist[j]=min{dist[j],dist[i]+matrix[i][j]}。根据这种思路，假设存在G=<V,E>，源顶点为V0，U={V0},dist[i]记录V0到i的最短距离，path[i]记录从V0到i路径上的i前面的一个顶点。

* 从V-U中选择使dist[i]值最小的顶点i，将i加入到U中；
* 更新与i直接相邻顶点的dist值。

(dist[j]=min{dist[j],dist[i]+matrix[i][j]})

* 直到U=V，停止。

|  |
| --- |
| //dijkstra算法  template**<**class VertexType**>** void MyGraph**<**VertexType**>::**ShortestPath\_Dijkstra**(**int sourcevertex**)**  **{**  int vex **=** sourcevertex**;**  //保存最短路径  int **\***ShortestPath **=** **new** int**[this->**mNumVertexes**];**  //保存起始点到各个顶点之间的权值和  int **\***TotalWeights **=** **new** int**[this->**mNumVertexes**];**  //保存顶点是否找到最短路径  bool **\***IsFindShortest **=** **new** bool**[this->**mNumVertexes**];**  //初始化上面三个数组  **for** **(**int i **=** 0**;** i **<** **this->**mNumVertexes**;** **++**i**){**  ShortestPath**[**i**]** **=** vex**;** //未找到路径，默认为vex  TotalWeights**[**i**]** **=** **this->**pEdges**[**vex**][**i**];** //每个顶点距离起始点的权值  IsFindShortest**[**i**]** **=** **false;** //每个顶点都还未找到  **}**  //自身顶点不需要找  IsFindShortest**[**vex**]** **=** **true;**  //自身到自身的距离为0  TotalWeights**[**vex**]** **=** 0**;**  //统计每一个顶点到vex的最短路径  **for** **(**int i **=** 0**;** i **<** **this->**mNumVertexes**;** **++**i**){**    //最小权值  int minWeights **=** N**;**  //距离最近的顶点  int lateseVextex **=** 0**;**  **for** **(**int j **=** 0**;** j **<** **this->**mNumVertexes**;** **++**j**){**  // TotalWeights[j] < minWeights 可保证从起始点开始寻找最短路径  **if** **(!**IsFindShortest**[**j**]** **&&** TotalWeights**[**j**]** **<** minWeights**){**  lateseVextex **=** j**;**  minWeights **=** TotalWeights**[**j**];**  **}**  **}**  //lateseVextex顶点已经被找到  IsFindShortest**[**lateseVextex**]** **=** **true;**  //现在我们除了起始点外，我们又知道了lateseVextex顶点，那么我们需要  //修正以下此刻存储权值和的表，也就是经过lateseVextex顶点之后，权值和  **for** **(**int k **=** 0**;** k **<** **this->**mNumVertexes**;** **++**k**){**    **if** **(!**IsFindShortest**[**k**]** **&&** **((**minWeights **+** pEdges**[**lateseVextex**][**k**])** **<** **(**TotalWeights**[**k**])))**  **{**  TotalWeights**[**k**]** **=** minWeights **+** pEdges**[**lateseVextex**][**k**];**  //记录下节点的前驱节点  ShortestPath**[**k**]** **=** lateseVextex**;**  **}**  **}**  **}**  **if** **(**ShortestPath **!=** **NULL){**  **delete[]** ShortestPath**;**  ShortestPath **=** **NULL;**  **}**  **if** **(**IsFindShortest **!=** **NULL){**  **delete[]** IsFindShortest**;**  IsFindShortest **=** **NULL;**  **}**  **if** **(**TotalWeights **!=** **NULL){**  **delete[]** TotalWeights**;**  TotalWeights **=** **NULL;**  **}**  **}** |

# 排序

## 4.1希尔排序

* 算法介绍

**希尔排序的实质就是分组插入排序**，该方法又称缩小增量排序，因DL．Shell于1959年提出而得名。

* 基本思想

**先将整个待排元素序列分割成若干个子序列（由相隔某个“增量”的元素组成的）分别进行直接插入排序，然后依次缩减增量再进行排序，待整个序列中的元素基本有序（增量足够小）时，再对全体元素进行一次直接插入排序。因为直接插入排序在元素基本有序的情况下（接近最好情况），效率是很高的，因此希尔排序在时间效率上比前三种方法有较大提高。**

|  |
| --- |
| #define MAX 10  void printArray**(**int arr**[],** int len**){**  **for** **(**int i **=** 0**;** i **<** len**;** i**++){**  printf**(**"%d "**,** arr**[**i**]);**  **}**  printf**(**"\n"**);**  **}**  //希尔排序  void ShellSort**(**int arr**[],**int len**){**    int increasement **=** len**;**  **do{**  //缩减增量  increasement **=** increasement **/** 3 **+** 1**;**  **for** **(**int i **=** 0**;** i **<** increasement**;** i **++){**  //遍历每一组  **for** **(**int j **=** i **+** increasement**;** j **<** len**;** j **+=** increasement**){**    **if** **(**arr**[**j**]** **<** arr**[**j **-** increasement**]){**    int temp **=** arr**[**j**];**  int k **=** j **-** increasement**;**  **for** **(;** k **>=** 0 **&&** temp **<** arr**[**k**];**k **-=** increasement**){**  arr**[**k **+** increasement**]** **=** arr**[**k**];**  **}**  arr**[**k **+** increasement**]** **=** temp**;**  **}**  **}**  **}**  **}** **while** **(**increasement **>** 1**);**  **}**  int main**(){**  int arr**[**MAX**];**  srand**((**unsigned int**)**time**(NULL));**  **for** **(**int i **=** 0**;** i **<** MAX**;** i**++){**  arr**[**i**]** **=** rand**()** **%** MAX**;**  **}**  //排序前  printArray**(**arr**,** MAX**);**  //冒泡排序  ShellSort**(**arr**,** MAX**);**  //排序后  printArray**(**arr**,** MAX**);**  system**(**"pause"**);**  **return** EXIT\_SUCCESS**;**  **}** |

通过这段代码，大家应该能够看到在希尔排序中我们并不是随便的把序列分组，然后对每个子序列分别排序，而是将相隔“增量”的记录组成一个子序列，实现跳跃式移动，使得排序效率提高。

这个增量的选取就非常关键了。我们案例中是用Increasement = increasement / 3 + 1,至于这个增量如何选取非常难，没有明确的确定，但是通过前人的研究，我们用

步长 = 步长 / 3 + 1。

## 4.2快速排序

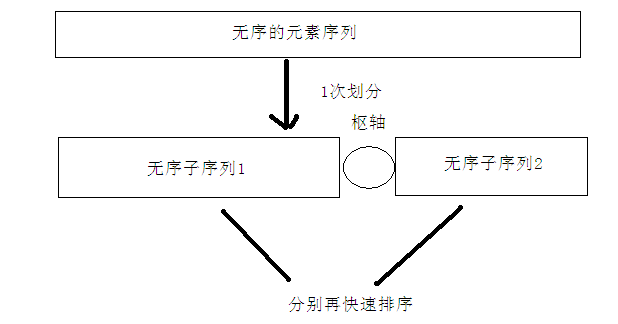
* 算法介绍

快速排序是C.R.A.Hoare于1962年提出的**一种划分交换排序**。它**采用了一种分治的策略**，通常称其为分治法(Divide-and-ConquerMethod)。快速排序被称为20世界十大算法之一。快速排序和冒泡排序类似，不断的通过比较和移动交换来实现排序，只不过他的实现增大了比较和移动的距离，把较大的数据移动到后面，较小的数据移动到前面，从而减少了了总的比较和移动交换次数。该算法是目前实践中使用最频繁、实践效率最好的排序算法。

* 分治法基本思想

通过一趟排序将待排序的数据分成两部分，其中一部分的数据均比另一部分小，然后再分别对这两部分进行排序，以达到整个序列有序的目的。

* + 先从数列中取出一个数作为基准数（枢轴）。
  + **分区过程将比这个数大的数全放到它的右边，小于或等于它的数全放到它的左边。(升序)**
  + 再对左右区间重复第二步，直到各区间只有一个数。



|  |
| --- |
| #define MAX 10  void printArray**(**int arr**[],** int len**){**  **for** **(**int i **=** 0**;** i **<** len**;** i**++){**  printf**(**"%d "**,** arr**[**i**]);**  **}**  printf**(**"\n"**);**  **}**  //快速排序  void QuickSort**(**int arr**[],**int start**,** int end**){**  int i **=** start**;**  int j **=** end**;**  //取基准数  int target **=** arr**[**start**];**  **if** **(**i **<** j**){**    **while** **(**i **<** j**){**    //从右向左找比基准数大的  **while** **(**i **<** j **&&** arr**[**j**]** **<** target**){**  j**--;**  **}**  **if** **(**i **<** j**){**  arr**[**i**]** **=** arr**[**j**];**  i**++;**  **}**  //从左向右找比基准数小的  **while** **(**i **<** j **&&** arr**[**i**]** **>** target**){**  i**++;**  **}**  **if** **(**i **<** j**){**  arr**[**j**]** **=** arr**[**i**];**  j**--;**  **}**  **}**    //i的位置就是基准数最合适的位置  arr**[**i**]** **=** target**;**  //快速排序左半部分  QuickSort**(**arr**,**start**,**i **-** 1**);**  //快速排序有半部分  QuickSort**(**arr**,** i **+** 1**,** end**);**  **}**  **}**  int main**(){**  int arr**[**MAX**];**  srand**((**unsigned int**)**time**(NULL));**  **for** **(**int i **=** 0**;** i **<** MAX**;** i**++){**  arr**[**i**]** **=** rand**()** **%** MAX**;**  **}**  //排序前  printArray**(**arr**,** MAX**);**  //冒泡排序  QuickSort**(**arr**,** 0**,**MAX **-** 1**);**  //排序后  printArray**(**arr**,** MAX**);**  system**(**"pause"**);**  **return** EXIT\_SUCCESS**;**  **}** |

## 4.3归并排序

* 算法介绍

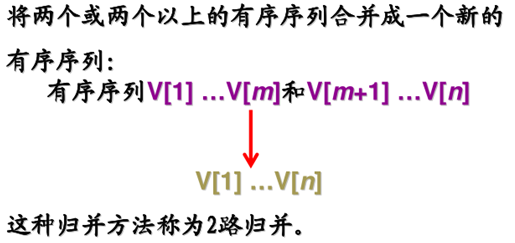
归并排序是建立在归并操作上的一种有效的排序算法。该算法是采用分治法（Divide and Conquer）的一个非常典型的应用。

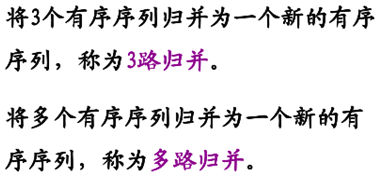
* 基本思想

**基本思路就是将数组分成二组A，B，如果这二组组内的数据都是有序的，那么就可以很方便的将这二组数据进行排序**。如何让这二组组内数据有序了？

**可以将A，B组各自再分成二组。依次类推，当分出来的小组只有一个数据时，可以认为这个小组组内已经达到了有序，然后再合并相邻的二个小组就可以了。这样通过先递归的分解数列，再合并数列就完成了归并排序。**

* **归并的定义**





* **如何合并连个有序序列？？？**

只要从比较二个数列的第一个数，谁小就先取谁，取了后就在对应数列中删除这个数。然后再进行比较，如果有数列为空，那直接将另一个数列的数据依次取出即可。

|  |
| --- |
| #define MAX 10  void printArray**(**int arr**[],** int len**){**  **for** **(**int i **=** 0**;** i **<** len**;** i**++){**  printf**(**"%d "**,** arr**[**i**]);**  **}**  printf**(**"\n"**);**  **}**  //合并两个有序序列  void Merge**(**int arr**[],** int start**,** int end**,** int mid**,** int tempSpace**[]){**    int iStart **=** start**;**  int iEnd **=** mid**;**  int jStart **=** mid**+**1**;**  int jEnd **=** end**;**  int length **=** 0**;**  **while** **(**iStart **<=** iEnd **&&** jStart **<=** jEnd**){**    **if** **(**arr**[**iStart**]** **<** arr**[**jStart**]){**  tempSpace**[**length**]** **=** arr**[**iStart**];**  iStart**++;**  **}**  **else{**  tempSpace**[**length**]** **=** arr**[**jStart**];**  jStart**++;**  **}**  length**++;**  **}**  //两个序列中有一个还有剩余元素  **while** **(**iStart **<=** iEnd**){**  tempSpace**[**length**]** **=** arr**[**iStart**];**  iStart**++;**  length**++;**  **}**  **while** **(**jStart **<=** jEnd**){**  tempSpace**[**length**]** **=** arr**[**jStart**];**  jStart**++;**  length**++;**  **}**  //覆盖原空间的数据  **for** **(**int i **=** 0**;** i **<** length**;**i**++){**  arr**[**start **+** i**]** **=** tempSpace**[**i**];**  **}**  **}**  //归并排序  void MergeSort**(**int arr**[],**int start**,**int end**,**int tempSpace**[]){**  **if** **(**start **==** end**){**  **return;**  **}**  int mid **=** **(**start **+** end**)** **/** 2**;**  //拆分左半部分  MergeSort**(**arr**,** start**,** mid**,** tempSpace**);**  //拆分右半部分  MergeSort**(**arr**,** mid **+** 1**,** end**,** tempSpace**);**  //拆分完了 合并有序序列  Merge**(**arr**,**start**,**end**,**mid**,**tempSpace**);**  **}**  int main**(){**  int arr**[**MAX**];**  srand**((**unsigned int**)**time**(NULL));**  **for** **(**int i **=** 0**;** i **<** MAX**;** i**++){**  arr**[**i**]** **=** rand**()** **%** MAX**;**  **}**  //排序前  printArray**(**arr**,** MAX**);**  //冒泡排序  int tempSpace**[**MAX**];**  MergeSort**(**arr**,** 0**,** MAX **-** 1**,** tempSpace**);**  //排序后  printArray**(**arr**,** MAX**);**  system**(**"pause"**);**  **return** EXIT\_SUCCESS**;**  **}** |

## 4.4排序总结

各种排序算法的比较需要考虑诸多的因素，在实际操作中，我们主要考虑时间复杂度和空间复杂度这两个方面。虽然仅仅靠这两个参数是无法判断一个算法的优劣，算法的实际性能还受运行的硬件、数据规模和数据特点的影响，但是这两个方面的比较对于算法的选择仍然具有很好的指导意义。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **排序算法** | **平均时间**  **复杂度** | **最坏时间**  **复杂度** | **平均空间**  **复杂度** | **稳定性** |
| 选择排序 | O(n2) | O(n2) | O(1) | 不稳定 |
| 冒泡排序 | O(n2) | O(n2) | O(1) | 稳定 |
| 插入排序 | O(n2) | O(n2) | O(1) | 稳定 |
| 希尔排序 | O(nlogn) | O(n2) | O(1) | 不稳定 |
| 快速排序 | O(nlogn) | O(n2) | O(logn) | 不稳定 |
| 归并排序 | O(nlogn) | O(nlogn) | O(n) | 稳定 |
| 堆排序 | O(nlogn) | O(nlogn) | O(1) | 不稳定 |

在了解了各种算法的理论性能之后，我们对算法的选择标准还要参考待排序数据的规模、存储空间的紧张程度和待排序数据的初识状态。综合考虑这些因素，我们给出以下几条建议：

* 在通常的情况下，即输出数据是随机的，快速排序、归并排序、希尔排序和堆排序的运行速度较快。其中堆排序是原地排序最省空间，快速排序的速度最快。所以如果在内存空间不紧张的情况下，一般采用快速排序，如果需要节省空间则采用堆排序，希尔排序不适合用在链表数据结构上。
* 如果待排序规模不大而且基本有序，插入排序和冒泡排序的运行时间很好，可选用这两排序算法。
* 从待排序的规模方面考虑，规模较小的时候，采用简单的排序算法比较合适，其整体性能比较高；如果规模较大，采用改进型的算法比较合适。

综合看来，每种算法都有其优缺点，很难说那种算法绝对比另一种算法好，我们需要根据具体情况具体分析，从而选择最合适项目的排序算法。