**第一章 算法与算法分析**

**§1.1 从问题到程序**

1. 搞清问题

\*要用程序解决问题，首先要搞清我们要解决什么问题。有些问题，比如：维护世界和平，无法用计算机解决。有人说，搞清了问题，问题就解决了一半了。

二． 用形式模形来描述问题

三．设计抽象数据类型

1. 什么是抽象数据类型？

抽象数据类型(ADT)是一个数学模型(通常有一个值的集合)以及定义在该模型上的一组操作。

\*例子：整数集合的集合以及集合的并、交、差运算。

一个群、环、域也是抽象数据类型。

\*面向对象的程序设计。

2. 数据类型、数据结构和抽象数据类型的差别

(1) 数据类型：在程序设计语言中，一个变量的数据类型是该变量所能取的值的集合。

\*例如：整型，布尔型

(2) 数据结构：具有不同数据类型的一组变量按各种形式联系起来。

(3) 抽象数据类型：是一个数学模型以及定义在该模型上的一组操作。

3. 实现抽象数据类型

\*根据实现的程序设计语言提供的数据类型，用各种数据结构组织起来实现。抽象数据类型的操作用程序的过程来实现。

4. 抽象数据类型的特点：

一般性

封装性

四. 设计算法

\*有了问题的数学模型和抽象数据类型后，可以设计解决问题的算法。

1. 什么是算法？

算法是一个有穷的指令序列。每条指令有清楚的含义，并且在有穷的时间内用有穷的动作能完成。一个算法无论接受任何输入都必须在有穷步内停止。

\*算法与程序不同，操作系统永远不停止，因此不是算法。而操作系统里的子任务是用算法实现的。

2. 关于指令

具体地说，在现实计算机上或图灵机上在有穷时间内能完成的才叫指令。

**§1.2 程序的运行时间**

1. 算法的设计目标
2. 算法应易于理解、编程和调试
3. 算法应尽可能有效地利用计算机的资源，特别地，它应尽可能快地运行。

\*两者矛盾，设计算法时需要权衡。

二. 程序运行时间的测量

影响程序运行时间的因素：

1. 程序输入的长度
2. 编译程序生成目标代码的质量
3. 计算机指令的性质和速度
4. 算法的时间复杂性

三．评价算法运行时间的标准

运行时间作为输入长度的函数T(n)

1. 最坏运行时间

算法对具有长度n的任何输入的最长运行时间

1. 最好运行时间

算法对具有长度n的任何输入的最短运行时间

1. 平均运行时间

即在“平均”输入下，算法的运行时间。通常我们假设给定长度的各种输入概率相同。平均运行时间是在这个假设下，运行时间的数学期望值。

1. 为什么常用最坏运行时间来估计

最坏运行时间是算法运行时间的上界，在实际问题中，算法的运行时间常常达到这个上界。平均运行时间难以计算。假定每一个输入具有相同的概率有时没有意义。平均运行时间常常与最坏运行时间有相同的数量级。

四．记号

1. 记号

设g(n)是一个给定函数，用表示函数的集合：

{存在正的常数和, 使得当, 有

}

我们写, 表示。

例1：

取, 当时，有。

2.记号O

设g(n)是一个给定函数，用表示函数的集合：

{存在正的常数和, 使得当, 有

}

我们写, 表示。

显然有 。

例2：同例1，有。同时，也有。

但是，不成立。

3.记号

设g(n)是一个给定函数，用表示函数的集合：

{存在正的常数和, 使得当, 有

}  
我们写, 表示。

定理1.1：对任意两个函数和, 当且仅当

且。

4.记号o

设g(n)是一个给定函数，用表示函数的集合：

{对任意正的常数c, 存在常数, 使得当, 有}

我们写, 表示。

例3：，但。

5.记号

设g(n)是一个给定函数，用表示函数的集合：

{对任意正的常数c, 存在常数, 使得当

, 有}

我们写，表示。

例4：, 但。

五．运行时间增长率的比较

例5：有两个算法运行时间分别为和，是否的算法比的算法好？

设, 。

当时，算法2比算法1运行得快，当充分大时，

，故当充分大时，算法1比算法2快。

例6：有4个算法，运行时间增长率如下：（见图1.1）

图1.1 图1.2

\*运行时间T(n)为n的多项式的算法称为好算法。

六．设计算法的几个原则：

1. 如果一个程序只用一两次，那么书写和调试所用的时间比程序运行时间大得多，因而算法应易于理解和正确实现。

2. 如果一个程序只对小的输入运行，运行时间增长率比运行时间前面的常数因子显得不重要。可选常数因子小而增长率大的算法。

3. 一个复杂而有效的算法可能不利于维护，有时应选择简单而相对低效的算法。

4. 时间复杂度小的算法，空间复杂度可能很大。

5. 对数值算法而言，精度和稳定性与时间高效一样重要。

**§1.3 算法分析技术**

一．计算程序运行时间

1. 加法规则：

如果和分别是两个程序段和的运行时间，，，那么程序段后跟的运行时间为

, 时间复杂度为。

这因为存在正的常数和，使得当时，；当时，。令，

, 当时，

2.乘法规则：

如果和为和，那么为

。

乘法规则主要用于循环结构的时间分析。

\*注意有，其中为常数。

二．例子：

PROCEDURE Bubblesort (VAR A : array [1n] of integer);

VAR

i, j, temp : integer;

BEGIN

1. FOR i := 1 TO n-1 DO
2. FOR j := n DOWNTO i+1 DO
3. IF A[j-1] > A[j] THEN

BEGIN

1. temp := A[j-1];
2. A[j-1] := A[j];
3. A[j] := temp;

END

END;

算法分析：

(4), (5), (6)句为。

(3)句取最坏情况，执行条件判断O(1), 语句内部O(1), 结果为O(1)。

(2)句执行循环控制条件为O(1)，内部O(1), 共循环次，

由乘法规则，时间为。

(1)句将各次循环的时间加起来，为

三．过程调用和GOTO语句的分析

\*假设没有递归调用，则把过程调用语句当作一个语句处理，被调用过程的执行时间即是该语句的执行时间。从而我们可以分析调用过程的执行时间。由于没有递归调用，按以上方法，我们最终可以得到主程序的运行时间。

GOTO语句只限于用在循环内某个位置跳出循环，异常终止循环。按最坏情况运行时间分析，可以假设不存在该GOTO语句。

**§1.4 递归程序的分析**

一．什么是递归程序？

\*一个过程在运行时直接或间接地调用自己，则该过程称为递归程序。

二．递归程序的分析

1. 例子：归并排序

以下过程Merge()将两个已排好序的表和归并成一个排好序的表L。

\*不妨设和中的元素按从小到大的顺序排列。具体做法是，取和中第1个元素，比较它们的大小，不妨设的第1个元素小，将其从中取出，放入L中。再取和中第1个元素比较，取小者放入L中。反复做上述工作，直到或中一个表的元素已取完，然后将另一个表剩下的元素放在L的末尾。

例7： 将以下元素归并排序

图1.3

算法：

FUNCTION Mergesort (L : LIST; n : integer) : LIST;

/\*L是长为n的表，返回排好序的表。设n是2的幂\*/

BEGIN

IF n=1 THEN RETURN(L)

ELSE BEGIN

把L分成两个长为n/2的表L1和L2;

RETURN(Merge(Mergesort(L1, n/2), Mergesort(L2, n/2)));

END

END;

2.列递推关系式

设T(n)是过程Mergesort在最坏情况下的运行时间，那么

\*上式的含义是：当时，该过程只需返回L，故只需要常数时间, 当时，该过程两次递归调用自身，且和的长度为, 故需时间，而Merge过程将和归并成一个排好序的表L，最坏需时间。从而。

3. 几点解释

(1) 只有当n为偶数时，(1.1)式才成立，另外，只有当n为2的幂时，才能得到解。

(2) 对任意n, 我们可以合理地假设，如果，。从而对2的幂解(1.1)式，可以对任意n给出一个合理的估计。

(3) 对奇数n，我们也可以把递归式中改为来解。这样使(1.1)式解起来复杂一些。

**§1.5 解递归方程**

一．解递归方程的三种途径

1.猜一个解，用递推关系式证明。

\*有时只猜的形式，将某些参数留待以后确定(例如：

未知)，当我们试图证对所有成立时，推出参数的值。

2.用递推关系式代入该式右边的T(m) (m < n)，直到所有项仅含T(1)。由于T(1)总是常数，我们可以把T(n)表是成n和常数的表达式。我们把这个表达式称为T(n)的闭式。

3.对某些常见类型的递推关系式求通解。

二．猜一个解

我们猜(1.1)式的解是。(以下如无特别说明，我们总是用表示)。将代入，不成立。可再试。

当时，由于(1.1)式给出的是的上确界，故有。

作归纳假设， (1.2)

对所有成立。我们试图证明

由于对，我们有

故

(1.3)

这里 。

令，由归纳法可证，对所有, 有

(1.4)

故 是。

三．扩展递推式

以(1.1)为例，由于, 故

(1.5)

从而

(1.6)

又由 代入(1.6)式，得

由归纳法，可得

(1.8)

假设 , 则由上式可写为

(1.9)

由于，并且，(1.9)式为

故 是。

**§1.6 一大类递归方程的通解**

一．一类常见的递归方程

\*递推关系式也叫递归方程

(1.10)

其中称为驱动函数。

在Mergesort过程的例子中，, 以为单位。

二．扩展递推式

由于

设, 则，那么

(1.11)

由于,

由于

故 。

三．齐次解与特解

(1.11)式中的称为(1.10)式的齐次解，称为(1.10)式的特解。

当驱动函数对所有成立时，齐次解就是递归方程(1.10)式的解。

齐次解是解所有子问题的代价。而特解则是建立子问题和把子问题的解合并在一起的代价。

要改进一个算法的时间复杂性，就要改进齐次解和特解中大的那一项所代表的算法中的那一部分。

四．乘法函数

1. 乘法函数的定义：

若函数满足：，则称为乘法函数，其中和是正整数。

例8：是乘法函数，因为。

2.当是乘法函数时，解(1.11)式。

当是乘法函数时，。

设，有

(1.12)

当时，有

(1.13)

讨论：

1. 当，(1.12)式为。其中。在此情形下，特解与齐次解相同。

2. 当，(1.12)式为，即。在此情形下，特解大于齐次解。在的特例下，。这时特解为或。

3. 当，(1.13)为。这时特解超过齐次解。在的特例下，(1.13)式为。

五．例子

例8：求以下递归方程的解

在所有情况下，。

解1：。

特解与齐次解相同，。

解3：。

特解大于齐次解，。

解2：。

特解大于齐次解，。

六．通解定理

定理1.2(通解定理): 设和为常数，是函数，并设由以下递归方程定义在非负整数上：

，

其中表示或。那么

1. 如果对某个常数成立，则；
2. 如果，则；
3. 如果对某个常数成立，并且对某个常数和所有充分大的成立，则。

证明：思路与前面的证明类似，略。

七．解递归方程的一些技巧与例子。

例9：

解：不是乘法函数，因为有系数2。是乘法函数。令

，则

由于，齐次解为，故为.

对于特解, 令, 不会影响特解。

因为，

故特解也是。

从而。因为，故。

利用通解定理，，

从而 。

例10：

解：这里，。

齐次解为，特解为：

因为 ，故为。

例11：

解：，

由通解定理，，其中，

故 。

例12：

解：变换递归方程的变量，令，递归方程为

再令 ，递归方程为

，这时，特解大于齐次解，

从而。

**第三章 集合的表示**

**§3.1 集合与集合上的抽象数据类型**

\*集合是所有数学分支的基础，在算法设计中，集合是许多重要的抽象数据类型的基础。

一．集合的定义：

1. 集合：集合是一组元素的整体。

2. 线性序：这里我们假设集合元素具有某种线性序。

即 , 中恰有一个成立；

若且，则有。

3. 集合的运算：

三．集合上的抽象数据类型

1. UNION(A, B, C):

2. INTERSECTION(A, B, C):

3. DIFFERENCE(A, B, C):

4. MERGE(A, B, C): 且

5. MEMBER(x, A): 若, 为真；若，为假。

6. MAKENULL(A): 置A为。

7. INSERT(x, A): 令

8. DELETE(x, A): 令

9. ASSIGN(A, B): 令

10. MIN(A): 返回A中最小元素。

11. EQUAL(A, B): 若, 为真；若, 为假。

12. FIND(x): 设有一组不相交的集合，FIND(x)返回包含x的那个集合。

**§3.2 集合的基本实现方法**

1. 位向量实现方法：

如果我们讨论的集合都是某一个小的全集的子集，该全集的元素为, 是某一给定数，那么我们可以用位向量来实现集合。元素就是向量的第位。

特别地，当小于或等于一个机器字的位数时，可用一个机器字实现位向量，位向量的运算可用逻辑运算实现。若相当大时，位向量可用布尔数组实现。

SET = packed array[1] of Boolean;

1. 用链表实现集合

每个链表表示一个集合，集合中元素就是表元。

如果用无序链表表示集合，INTERSECTION操作需用时间，n为集合A和B的元素个数。改用有序链表，则只需时间。但无序链表INSERT操作只需时间，而有序链表INSERT操作需时间。MEMBER操作在最坏情况下需时间，平均也需时间。

用链表实现集合比较低效，但链表可用来表示任意的集合，实现起来也简单。

**§3.3 字典和优先队列**

一．字典

1. 什么是字典？

字典是一个集合抽象数据类型，该集合上只有操作INSERT, DELETE和MEMBER。(字典中也使用MAKENULL操作，将数据结构初始化)。

2.字典的实现

字典可用无序或有序链表实现。如果集合元素的全集是一个小的集合，也可用位向量实现。

二．优先队列

1. 什么是优先队列？

优先队列是一个集合抽象数据类型，该集合上只有操作INSERT和DELETEMIN(也包括MAKENULL)，其中DELETEMIN(S)是删除集合S中的最小元。

2.优先队列的实现

用无序链表实现，INSERT需时间，但DELETEMIN需时间。

用有序链表实现，INSERT需时间，而DELETEMIN只需时间。

优现队列还可以用堆来实现。建堆需时间，而以后每次执行INSERT和DELETEMIN只需时间。对于堆的详细讨论，见后面“堆排序”一节。

**§3.4 带有MERGE和FIND操作的集合**

一．背景：

在某些问题中，我们有一组对象，每一个对象属于某一个集合，这些集合两两不相交。我们需要求这些集合的并，以及求每一个对象所属的集合。

这些问题可用MERGE和FIND操作来实现。

例：集合上等价关系划分的等价类。

设，求在以下等价关系下，集合的等价类：, 。等价类为：。

\*从初始等价类：构造。

二．MFSET的简单实现

MFSET是一个抽象数据类型ADT，它含有一组子集，这些子集称为分支，并含有以下操作：

1. MERGE(A, B): 求不相交的集合A和B的并。
2. FIND(x): 返回x所属的子集。
3. INITIAL(A, x): 建立一个称为A的分支，其中只含一个元素x。
4. MFSET的数据结构：

CONST

n={number of elements}

TYPE

MFSET=array [1n] of integer;

更一般地

MFSET=array [subrange of members] of (type of set names)

\*每个数组元素的下标代表一个集合元素，该数组元素的值代表该集合元素所属的集合。

2. 操作：

(1) INITIAL(A, x): 令C : MFSET的元素C[x] := A;

(2) FIND(x): 设 C : MFSET，返回C[x];返回下标对应元素所属集合

(3) MERGE

PROCEDURE MERGE (A, B : integer; VAR C : MFSET);

VAR

x : 1n;

BEGIN

FOR x := 1 TO n DO

IF C[x] = B THEN C[x] := A;

END;

三．MFSET的一个快速实现

1. 简单实现的时间分析

INITIAL和FIND只需时间。

执行一系列个MERGE操作，最坏情况下需时间。

合并A和B，假设规定扫描完B中所有元素。设A中只有一个元素，第次执行MERGE操作，B中有个元素, 执行次MERGE操作，需时间。

1. 改进方法

合并两个集合A和B，扫描元素少的那个集合，将它合并到大的集合中。

这样每次合并，集合元素个数(比小的集合)至少增长一倍。假设开始有n个集合，则n个元素最多改变次它所属的分支，故改变的总数至多为。即为时间。

1. 数据结构

TYPE

nametype = 1n;

elementtype = 1n;

MFSET = record

setheaders : array [1n] of record

count : 0n;

firstelement : 0n;

end;

names : array [1n] of record

setname : nametype;

nextelement : 0n;

end

end

1. 算法：

PROCEDURE INITIAL (A : nametype; x : elementtype; VAR C : MFSET);

BEGIN

C.names[x].setname := A;

C.names[x].nextelement := 0;

C.setheaders[A].count := 1;

C.setheaders[A].firstelement := x;

END;

PROCEDURE MERGE (A, B : nametype; VAR C : MFSET);

VAR

i : n;

BEGIN

IF C.setheaders[A].count > C.setheaders[B].count THEN

BEGIN

i := C.setheaders[B].firstelement;

WHILE C.names[ i ].nextelement 0 DO

BEGIN

C.names[ i ].setname := A;

i := C.names[ i ].nextelement;

END;

C.names[ i ].setname := A;

C.names[ i ].nextelement := C.setheaders[A].firstelement;

C.setheaders[A].firstelement := C.setheaders[B].firstelement;

C.setheaders[A].count := C.setheaders[A].count +

C.setheaders[B].count;

C.setheaders[B].count := 0;

C.setheaders[B].firstelement := 0;

END

ELSE /\* B is at least as large as A \*/

/\* code similar to case above, but with A and B interchanged \*/

END;

FUNCTION FIND (x : 1n; VAR C : MFSET);

BEGIN

RETURN( C.names[x].setnames )

END;

四．用树实现MFSET

\*本节我们用非形式化的方法讲明这种实现途径的思想。

1. 思想：

每个子集用一棵树来表示。树的每个结点对应一个集合元素，每个元素通过一个映射与它的结点对应起来。树中每个结点有一个指针指向它的父结点(根结点除外)。根结点中保存该集合的名字。

2.操作

(1) FIND(x): 用映射找到x对应的树结点，然后沿着该结点到根的路径找到根，即得本集合的名字。

(2) MERGE(A, B): 将A和B中元素个数少的那个集合(不妨设为B)的树根作为A的树根的孩子。

3. 算法分析：

FIND操作需时间，MERGE操作需时间。

**§3.5 Hash表**

\*用链表来实现字典，每一个INSERT, DELETE或MEMBER操作，平均时间为O(n)，n为字典中元素个数。而散列(hashing)技术实现字典，以上每个操作平均时间为O(1)。

一．什么是散列技术

Hash(散列)表技术是基于建立从关键字到记录存储地址之间的函数(映射)关系而进行查找的。

设关键字的全集为U。字典存放在长为m的Hash表中。我们建立Hash函数，对每个元素的关键字k，称为k的hash函数值。Hash技术的基本思想是用作为k在hash表中的存储地址。当有两个不同的关键字和有相同的hash函数值时，即，则产生冲突。

二．链地址法

1. 方法：

链地址法是为hash表T的每一个表元建立一个链表，即为链表的表头。所有hash函数值为的元素存放在为表头的链表中。它们的插入、删除和查找均对该链表进行。

1. 链地址法的分析

设hash表长为m(m个入口)，存放了n个元素。称为装载因子。

1. 最坏情况

所有元素的hash值相同，它们存在hash表的同一个入口的链表中。最坏情况下查找需时间。

1. 平均时间

我们假设任一关键字的hash值落在T的m个入口的可能性相同。这个假设称为简单均匀散列。

定理3.1：在简单均匀散列的假设下，链地址法不成功的查找平均时间为。

证明：在简单均匀散列的假设下，任一关键字k的hash值落在m个入口的可能性相同。查找关键字k的不成功查找的平均时间是查找m条链中的一条从头到尾所需的查找时间。每条链的平均长度为，而计算h需时间，故所需总时间为。

定理3.2：在简单均匀散列的假设下，用链地址法的成功查找所需平均时间为。

证明：假设hash表中n个元素被查找的可能性相同。假设插入元素时，插入在每一条链的尾部。在成功的查找中，平均访问元素的个数是被查找的元素插入时访问的元素个数加1。插入第个元素时，每条链平均长度为。故成功查找访问元素的平均次数为

故成功查找平均时间(包括计算hash函数)为=

。

三．Hash函数

1.好的hash函数的标准

就是要满足简单均匀散列假设，即每个关键字的hash值落在m个入口的可能性相同。

在计算hash值时，我们应把每个关键字解释成一个自然数。

2.除方法

对任一关键字k， 。

如果关键字被表示成p进制数，m不宜取p的幂。m常取的值为接近2的幂的某个素数。

3.乘方法

首先用关键字k乘某个常数A，, 再取kA的小数部分乘m下取整。表达为

, 其中，表示。

假设一个机器字为w位，k占一个机器字。用k乘一个w位的整数，结果为2w位的数，取的p位有效位作为。

其中，。

一般 较好。

4.全局散列

在最坏情况下，所有关键字的散列值可能相同，查找需时间。

我们选择一个精心设计的hash函数的集合H, 然后独立于关键字随机地从H中选一个hash函数，作为我们所用的hash函数。这种方法称为全局散列。H称为是全局的，是说：对每一对不同的关键字，满足的H中的hash函数h恰有|H|/m个。其中，H, 。

从理论上可以证明，全局散列有希望得到较好的平均时间复杂性。

四．开放定址法

1. 方法：

在开放定址法中，所有元素都存在hash表中，每个表元或者保存一个元素，或者是NIL。当查找关键字k时，计算h(k)，若T[h(k)]已有元素，但不是k, 则我们按某种方式，求T中下一个位置，继续查找。下一个位置如果已有元素而且不是k, 则再求下一个位置，这个过程称为探测。在用开放定址法进行插入时，反复探测，直到找到一个空表元，插入相应的关键字。

Hash函数：

。

对每个关键字k, 探测序列

是的一个排列。

2.操作：

PROCEDURE Hash\_Insert (T, k);

BEGIN

i := 0;

REPEAT

j := h(k, i);

IF T[j] = NIL THEN

BEGIN

T[j] := k;

RETURN(j);

END

ELSE i := i+1

UNTIL i = m;

ERROR(“hash table overflow”);

END;

PROCEDURE Hash\_Search (T, k)

BEGIN

i := 0;

REPEAT

j := h(k, i);

IF T[j] = k THEN RETURN(j);

i := i+1;

UNTIL (T[j] = NIL) OR (i = m);

RETURN(NIL);

END;

当用DELETE删除一个元素时，该元素被赋予一个特殊值deleted。查找时，把deleted当作一个元素处理，而插入时，把deleted当作NIL处理。

1. 线性探测

设是通常的hash函数，，

。

1. 平方探测

，

其中，是辅助hash函数(同上), 为常数。

1. 双重散列

，

其中，和是两个辅助hash函数。

通常要求的值与hash表的长度m互素。假若不然，与m的最大公因子，则查找关键字k时，只查找了hash表的个表元。

例如：我们可以选

比稍微小一点(比如：或)。

1. 开放定址法的分析

在以下分析中，我们总是假设：对每一个关键字，它的探测序列取的每一种排列的可能性相同。这称为均匀散列假设。当k确定后，它的探测序列也就确定了。

定理3.3：在均匀散列假设下，给定具有装载因子的开放定址hash表，不成功查找的平均探测数为。

证明：在不成功查找中，除最后一次探测外，每次探测都访问一个不是预期的元素，最后一次访问一个空表元。定义

恰好次探测被占用的表元的概率，

对，，因为至多有个表元被占用。

那么平均探测数为

(3.1)

为了计算(3.1)式，定义

至少次探测被占用的表元的概率，

由概率论的知识，有

我们有

。当时，

。

推论3.4：在均匀散列的假设下，给定装载因子为的开放定址hash表，在该表中插入一个元素平均需要次探测。

证明：仅当该表中有空位时，才能插入元素，这时，。插入一个元素时，首先进行一次不成功的查找，然后在所找到的第一个空位处插入元素。故平均探测数为。

定理3.5：假设是均匀散列，并且hash表中每一个关键字被查找的可能性相同。给定装载因子为，则一次成功查找的平均探测数为

证明：查找关键字k的探测序列与插入关键字k的探测序列相同。由推论3.4，如果k是第个被插入的元素，查找k的平均探测数为。成功查找中，个关键字的平均探测数为

其中，。可以证明 。

因而有

因而本定理得证。

**第四章 实现集合的高级方法**

**§4.1 二叉查找树**

\*假设集合有线性序，用二叉查找树表示集合，INSERT, DELETE, MEMBER和MIN操作平均时间复杂度为。

一．二叉查找树

1. 定义：一个二叉查找树是一棵二叉树，用集合元素标记每个结点。每个结点的左子树上的元素都小于该结点上的元素，每个结点右子树上的元素都大于该结点上的元素。

2. 例子：

图4.1

二．二叉查找树的操作

1. 数据结构

type

nodetype=RECORD

element : elementtype;

leftchild : nodetype;

rightchild : nodetype;

END;

SET=nodetype;

2. 操作

FUNCTION MEMBER (x : elementtype; A : SET) : Boolean;

/\*returns true if x is in A, false otherwise\*/

BEGIN

IF A = nil THEN

RETURN(false) /\*x is never in \*/

ELSE IF x = A.element THEN

RETURN(true)

ELSE IF x < A.element THEN

RETURN( MEMBER(x, Aleftchild) )

ELSE /\* x > A.element \*/

RETURN( MEMBER(x, Arightchild) )

END;

PROCEDURE INSERT (x : elementtype; VAR A : SET);

/\* add x to set A \*/

BEGIN

IF A = nil THEN

BEGIN

new(A);

A.element := x;

A.leftchild := nil;

A.rightchild := nil;

END

ELSE IF x < A.element THEN

INSERT(x, A.leftchild)

ELSE IF x > A.element THEN

INSERT(x, A.rightchild);

/\* if x = A.element, we do nothing; x is already in the set \*/

END;

FUNCTION DELETEMIN (VAR A : SET) : elementtype;

/\* returns and removes the smallest element from set A \*/

BEGIN

IF A.leftchild = nil THEN

BEGIN /\* A points to the smallest element from set A \*/

DELETEMIN := A.element;

A := A.rightchild;

/\* replace the node pointed to by A by its right child \*/

END

ELSE /\* the node pointed by A has a leftchild \*/

DELETEMIN := DELETEMIN(A.leftchild)

END;

PROCEDURE DELETE (x : elementtype; VAR A : SET);

/\* removes x from set A \*/

BEGIN

IF A <> nil THEN

IF x < A.element THEN

DELETE(x, A.leftchild)

ELSE IF x > A.element THEN

DELETE(x, A.rightchild)

/\* if we reach here, x is at the node pointed by A \*/

ELSE IF (A.leftchild = nil) AND (A.rightchild = nil) THEN

A := nil /\* delete the leaf holding x \*/

ELSE IF A.leftchild = nil THEN

A := A.rightchild

ELSE IF A.rightchild = nil THEN

A := A.leftchild

ELSE /\* both children are present \*/

A.element := DELETEMIN(A.rightchild)

END;

三．二叉查找树操作的时间复杂性

1. 最坏情况

当输入元素是有序的(从小到大)，则二叉查找树退化为一个链表，各种操作的时间复杂度同链表表示法。

例入：MEMBER操作需O(n)时间，插入第个元素需时间，插入n个元素需

时间。

2. 平均时间复杂性

若二叉树是完全树，从根结点到叶子的路径上结点数为，故MEMBER, INSERT, DELETE, DELETEMIN只需时间。

下面证明，在随机的情况下建立二叉查找树，从根结点到任一结点的路径上平均结点数为。假设：所建立的二叉查找树只是由插入操作建立的，插入时n个结点的各种排列顺序的可能性相同。令P(n)为n个结点的二叉查找树从根结点到任一结点的路径的平均结点数。则。

设是输入二叉查找树的第一个结点，有个结点小于，有个结点大于。

由归纳假设，的左子树从根到任一结点路径平均结点数为，的右子树从根到任一结点路径平均结点数为。

当左子树有个结点，右子树有个结点时，从根到任一结点路径的结点数为

下面归纳证明： 。

当时，。

对任意，设 。

故

当时，

当时，

故MEMBER, INSERT, DELETE, DELETEMIN, MIN操作平均时间复杂度均为。

**§4.2 红黑树**

从上一节的内容可知，给定一个高度为h的二叉查找树，其上的基本动态集合操作，如：MEMBER, PREDECESSOR, SUCCESSOR, MINIMUM, MAXIMUM, INSERT, DELETE可以在时间实现。而二叉查找树的平均高度为。**这一节讲的红黑树将保证以上操作在最坏情况下，执行时间为。**

一．红黑树的性质

1. 什么是红黑树？

**一棵红黑树是一棵二叉查找树，**每个结点有一个域保存它的颜色，该颜色或者是红，或者是黑。**在红黑树中，从根到任一叶子的路径长度至多是从根到另一叶子的路径长度的两倍。**

2. 红黑树结点的数据结构

红黑树的每个结点是一个记录，包括以下域：color, key, left, right, p。其中，left和right分别指向该结点的左、右孩子。p指向该结点的父亲结点。

**在红黑树中，若某个结点没有左(右)孩子，则它的left(right)指针指向一个关键字为NIL的叶结点。NIL结点具有与普通结点相同的结构。**

3. 红黑树的性质

(1) 每一个结点或者是红，或者是黑；

(2) **每个叶子(NIL)是黑**；

(3) **如果一个结点是红，则它的两个孩子都是黑；**

**(4) 从一个结点到它任一后代叶结点的路径具有相同数目的黑结点。**

**例：**

图4.2

**设某个结点到后代叶结点路上的黑结点数(不含)称为的黑高度，记为。**

引理4.1：有n个内部结点的红黑树的高至多为。

证明：首先证明以结点为根的子树含至少个内部结点。对的高度用归纳法证明。如果的高度是0，则必然是叶子(NIL)，因而以为根的子树含至少个内部结点。归纳步，考虑一个具有正的高度的结点，并且具有两个孩子。则的每个孩子如果是红色，则其黑高度为bh(), 如果是黑色，则其黑高度为。由归纳假设，的每个孩子为根的子树至少有个内部结点，故为根的子树有至少()+()+1

个内部结点。

设h是树的高度，由性质(3)，从根结点到叶子结点的任一路径上至少一半结点(不含根)是黑色，因而树的黑高度至少为。所以

由上式变换可得，故。

4. 算法分析

由引理4.1知，在红黑树上，MEMBER, MINIMUM, MAXIMUM, SUCCESSOR, PREDECESSOR可在时间内实现。

INSERT, DELETE可能破坏红黑树的性质。下面我们将提供在时间内实现INSERT和DELETE的算法。

二．旋转操作

在二叉查找树的INSERT和DELETE操作中，操作结果可能与红黑树的性质相冲突，为此我们引入旋转操作，用以调整二叉查找树，使其保持红黑树的性质。

1. 旋转操作

图4.3

1. 算法：

PROCEDURE LEFT\_ROTATE(T, x);【x左旋转】

BEGIN

y := x.right;

x.right := y.left;

IF y.left NIL THEN

y.left.p := x;

y.p := x.p;

IF x.p = NIL THEN

root[T] := y

ELSE IF x = x.p.left THEN

x.p.left := y

ELSE x.p.right := y;

y.left := x;

x.p := y;

END;

RIGHT\_ROTATE与上述过程相似。

1. 算法分析

**LEFT\_ROTATE和RIGHT\_ROTATE操作只需O(1)时间。**

三．插入操作

以下给出红黑树插入元素的操作，它首先按二叉查找树的方法插入一个元素x，并给x红色，然后调整整棵红黑树使其保持红黑树的性质。

1. 算法：

PROCEDURE RB\_INSERT (T, x);

BEGIN

INSERT(T, x)

x.color := RED;

WHILE (x root[T]) AND (x.p.color = RED) DO

IF root[T] = x.p THEN

x := root[T]

ELSE IF x.p = x.p.p.left THEN

BEGIN

y := x.p.p.right;

IF y.color = RED THEN

BEGIN Case 1

x.p.color := BLACK;

y.color := BLACK;

x.p.p.color := RED;

x := x.p.p;

END

ELSE BEGIN

IF x = x.p.right THEN

BEGIN Case 2

x := x.p;

LEFT\_ROTATE(T, x);

END;

x.p.color := BLACK; Case 3

x.p.p.color := RED; Case 3

RIGHT\_ROTATE(T, x.p.p); Case 3

END

END

ELSE /\*same as THEN clause with “right” and “left”

exchanged \*/

root[T].color := BLACK;

END;

1. 算法分析

n个结点的红黑树高为。调用INSERT过程需时间，WHILE循环只有在执行Case 1时，才重复执行，指针x沿树中路径向根运动，因此WHILE 循环执行总次数为。故

RB\_INSERT需时间。

四．删除操作

红黑树的删除操作与插入操作类似，先按二叉查找树的方法删除一个元素，然后调整红黑树，使该树满足红黑树的性质。

每个叶子是一个标记为NIL的结点。为了节省存储空间，可用一个特殊的NIL结点代表所有的叶子。但在算法中必须小心地设置它的p指针域。

1. 算法：

PROCEDURE RB\_DELETE (T, z);

BEGIN

IF (z.left = nil[T]) OR (z.right = nil[T]) THEN y := z

ELSE y := SUCCESSOR(z);(z的后继)

IF y.left nil[T] THEN x := y.left

ELSE x := y.right;

x.p := y.p;

IF y.p = nil[T] THEN root[T] := x

ELSE IF y = y.p.left THEN y.p.left := x

ELSE y.p.right := x;

IF y z THEN z.key := y.key;【y是z的后继】

IF y.color = BLACK THEN

RB\_DELETE\_FIXUP(T, x);

RETURN(y);（如果y是红色的，删掉y并不影响树的结构）

END;

PROCEDURE RB\_DELETE\_FIXUP (T, x);【更新红黑树结构】

BEGIN

WHILE (x root[T]) AND (x.color = BLACK) DO

**IF** x = x.p.left THEN（x是一个左孩子）

**BEGIN**

w := x.p.right;（令w是另一边的右孩子）

**IF** w.color = RED THEN（如果w是红色的）

**BEGIN**  Case 1

w.color := BLACK;

x.p.color := RED;

LEFT\_ROTATE (T, x.p);

w := x.p.right;

END;

**IF** (w.left.color = BLACK) AND (w.right.color = BLACK) THEN BEGIN Case 2

w.color := RED;

x := x.p;

END

**ELSE** BEGIN

IF w.right.color = BLACK THEN

BEGIN Case 3

w.left.color := BLACK;

w.color := RED;

RIGHT\_ROTATE (T, w);

w := x.p.right;

END;

w.color := x.p.color; Case 4

x.p.color := BLACK; Case 4

w.right.color := BLACK; Case 4

LEFT\_ROTATE (T, x.p); Case 4

x := root[T]; Case 4

END

END

ELSE /\*same as THEN clause with “right” and “left” exchanged\*/

x.color := BLACK;

END;

1. 算法分析：

由于红黑树的高为, RB\_DELETE不含RB\_DELETE\_FIXUP调用所花的时间为。在RB\_DELETE\_FIXUP中，只有在Case 2下，WHILE循环才重复执行，指针x沿树中路径向根移动，因而WHILE循环最多重复次，而整个过程的执行时间为。故删除操作可在时间内完成。

**第五章 分类与查找**

**§5.1 基于比较排序算法的时间下界**

一．基于比较排序算法举例

插入排序

PROCEDURE Insertion\_Sort (A);

BEGIN

A[0].key := ;

FOR i := 2 TO n DO

BEGIN

j := i;

WHILE A[ j ].key < A[ j-1 ].key DO

BEGIN

swap(A[ j ], A[ j-1]);

j := j-1;

END

END

END;

二．判定树

基于比较的排序算法，其过程可用一个判定树来描述。

例：插入排序，A[1], A[2], A[3]中存放元素，排序过程的判定树见图5.1。

判定树是一棵二叉树，其每个结点代表算法比较元素后得到的状态。

图5.1

三．基于比较排序算法的时间下界

\*要点：

要将n个元素排序，n个元素的n!种排列都是可能的。

判定树至少有n!个叶子，因为每一种可能的排序是一个叶子。

删掉对应不必要的比较的结点，删掉对应无序的叶子(前面的比较不相容)，恰有n!个叶子。

判定树从根结点到叶子的最长路径的结点数是最坏情况下，排序所需比较次数的下界。

定理5.1：用基于比较的方法把n个元素排序至少需要次比较，其中为某个大于0的常数。

证明：首先证明：高度为h的二叉树有至多个叶子。

当时，二叉树只有根结点，恰好有个叶子。

设当时()，二叉树有**至多个（n!）叶子**。当时，根结点有左、右两个子树，每个子树的高度小于等于。于是二叉树的叶子数。

再证任何将n个不同元素排序的判定树**高度至少为** 。

因为将n个元素排序的结果可以是输入的n个元素的n!个排列中任何一个，故判定树至少有n!个叶子。由上面的结论，它的高度至少为。

最后证明本定理的结论。由于，

因此，。证毕。

结论：

1. 基于比较排序算法最坏情况下时间复杂度为。
2. 基于比较排序算法平均时间复杂度为。

**§5.2 堆排序**

一．例子：

将下列元素建成堆并排序

从最后一个有孩子的开始，一直做pushdown（把孩子里最小的换上来），之后更新指针指向其前一个元素（注意不是父亲），再接着继续做pushdown。使得堆的顶部为最小元。之后将最小元和最后一个元素互换值，之后从顶开始pushdown到n-1个元素。使得最后的集合是从大到小的排序。

图5.2

二．算法：

PROCEDURE Pushdown (first, last : integer);

VAR

r : integer;

BEGIN

r := first;

WHILE r last DIV 2 DO

IF 2\*r = last THEN

BEGIN

IF A[r].key > A[2\*r].key THEN

swap(A[r], A[2\*r]);

r := last

END

ELSE IF (A[r].key > A[2\*r].key) AND 【last>2r】

(A[2\*r].key A[2\*r+1].key) THEN

BEGIN

swap(A[r], A[2\*r]);

r := 2\*r;

END

ELSE IF (A[r].key > A[2\*r+1].key) AND

(A[2\*r+1].key < A[2\*r].key) THEN

BEGIN

swap(A[r], A[2\*r+1]);

r := 2\*r+1

END

ELSE r := last【父亲不大于小儿子】

END;

PROCEDURE Heapsort (A);

/\*将A[1], A[2], , A[n]排序\*/

VAR

i : integer;

BEGIN

1. FOR i := n DIV 2 DOWNTO 1 DO
2. Pushdown (i, n);【建堆】

(3) FOR i := n DOWNTO 2 DO【第一个元素和最后一的元素互换】

BEGIN

(4) swap(A[1], A[i]);

(5) Pushdown (1, i-1);

END

END;

三．算法分析：

1. 最坏情况下时间复杂度：

\* Pushdown过程：

每次做WHILE循环比较两次，

r初值为first, 经过i次循环，r first\*，

最后r > last/2 , 即first\* > last/2，

即 。

WHILE循环次数。

因为且，

故循环次数，比较次数。

Pushdown过程时间复杂度为。

\*Heapsort过程：

(1)-(2)行：时间复杂度为

(3)-(4)行：时间复杂度为

(3)-(5)行：时间复杂度为

总的时间复杂度：。

2. 平均时间复杂度：。

**§5.3 快速排序**

一．例子：将以下序列排序：3, 1, 4, 1, 5, 9, 2, 6, 5, 3

\***取前两个不同元素的大者作标杆**

左指针从左向右扫描，右指针从右向左扫描，当左边标杆，右边标杆时，交换左、右元素，继续上述过程，直到左指针到右指针的右边。

分成两组之后，左指针左边的是小于标杆的，右边是大于等于标杆的，之后递归这两组

二．算法：

FUNCTION Findpivot (i, j : integer) : integer;（返回标杆的下标）

VAR

firstkey : keytype;

k : integer;

BEGIN

firstkey := A[i].key;

FOR k := i + 1 TO j DO

IF A[k].key > firstkey THEN

RETURN(k)

ELSE IF A[k].key < firstkey THEN

RETURN( i );

RETURN(0);

END;

FUNCTION Partition (i, j : integer; pivot : keytype) : integer;

/\*将A[ i ], , A[ j ]分成两组，左边一组的关键字均小于标杆，右边一组的关键字均大于等于标杆，返回右边一组的第一个元素的下标\*/

VAR

l, r : integer;

BEGIN

l := i ; r := j ;

WHILE l r DO

BEGIN

WHILE A[ l ].key < pivot DO

l := l +1;

WHILE A[ r ].key pivot DO

r := r 1;

IF l < r THEN swap(A[ l ], A[ r ]);

END;

RETURN( l );

END;

PROCEDURE Quicksort (i, j : integer);

/\* 将排序 \*/

VAR

pivot : keytype;

pivotindex : integer;

k : integer;

BEGIN

1. pivotindex := Findpivot(i, j);
2. IF pivotindex 0 THEN

BEGIN

1. pivot := A[pivotindex].key;
2. k := Partition (i, j, pivot);
3. Quicksort (i, k-1);
4. Quicksort (k, j);6

END

END;

三．最坏运行时间

1. Findpivot : 最坏情况下：；

2. Partition：最坏情况下：；

游标指向到中每一个元素时，所花的时间不超过某一个常数，而游标决不会再回到其中任一元素，因为当时，该过程停止。故执行时间不超过。

3. Quicksort

(1) ：

(2), (3) :

(4) :

每次递归调用的时间与该调用要排序的元素个数成比例。总的运行时间等于所有递归调用中要排序的元素个数之和，也等于每个元素出现在不同递归调用中的次数之和。即在一开始例子中，每个元素排好序时所在的层数求和。

在最坏情况下，处理一个已排好序从小到大的数组(见图5.3)。

的层数为，的层数为n，各元素深度之和为：

故最坏情况下，需时间。

四．平均运行时间：

假设中元素各不相同。输入元素的各种顺序概率相同。递归调用时，元素的各种顺序概率也相同，标杆出现在排好序的数组中第2个，第3个，，第个位置的概率相同。

假设标杆出现在第个位置上，Partition后，左边有i个元素，右边有(n-i)个元素。

根据选标杆的方法，或者标杆在第一个位置，比标杆小的i个元素之一在第二个位置；或者比标杆小的i个元素之一在第一个位置, 标杆在第二个位置。

标杆在第一个位置的概率：

比标杆小的i个元素之一在第二个位置的概率：

第一种情况的概率：

同理，第二种情况的概率：

设对个元素排序的平均时间为。

T(1)

上式中为将个元素Partition所需的时间。

因为 (5.2)

故 (5.3)

(5.4)

(5.5)

要解(5.5)式，我们猜测一个结果 ， 然后用归纳法证明。

时，

设时，

(5.6)

(5.7)

取，则。

**故平均时间复杂度为。**

**\*快速排序平均时间比堆排序平均时间少一个常数因子，因而平均时间最少。**

**§5.4 基数排序及其应用**

\*5.1节给出了基于元素比较的排序算法的时间下界。当关键字的类型具有某些规律时，排序的时间可以少于。

一．思想：基数排序是不基于元素比较的排序方法，它是按组成关键字各位的值来实现排序的。

关键字可以看作是一个k元组其中每个的值域为0至。关键字的大小用字典序排序。即

当且仅当以下两个条件之一成立：

1.存在整数j，使得，而对所有;

2.且对任意。

二．算法：

PROCEDURE Radixsort;

/\*输入序列，其中是k元组，的值域为0至。输出序列，它是的某个排列，满足\*/

BEGIN

把依次放入队列QUEUE;

FOR j := k DOWNTO 1 DO

BEGIN

FOR l := 0 TO DO

将Q[l]置为空队列；

WHILE QUEUE非空 DO

BEGIN

设是QUEUE中第一个元素；

将从QUEUE移到队列尾；

END;

FOR l := 0 TO DO

将队列的内容接到QUEUE队列尾；

END

END;

三．算法分析：

上述算法在最坏情况下时间复杂度为。最外层循环每执行一次需时间，共循环k次。

四．不等长的关键字的基数排序

思想：本算法对n个不等长的串进行排序，每个串中的每一个成份的值域为0至。第i个串的长度为，。

算法的本质是先将n个串按串长递减的顺序排列。设是最长串的长度。用基数排序，先对长为的串的第位排序，然后对长度大于等于的串的第位进行排序，依此下去。

五．排序方法

输入：一系列串(元组)，每个串的每一个成份的值域为0至。且的长度，设。

输出：的一个排列, 满足：。

1.首先，对的每个l，建立一个链表，该链表中的元素是出现在串中第l个成份。我们首先对的每个i和l，构造有序对，然后用基数排序将所有这些偶对按字典序排序，建立一个表。从左到右扫描这个表，建立个有序链表

NONEMPTY[l] 。NONEMPTY[l]按顺序包含整数j，使得

对某个i成立。

2.确定每个串长，然后对，建立链表LENGTH[l], LENGTH[l]中包含所有长为l的串。

3.用基数排序对串的每个成份排序，从第位开始，第i遍扫描之后，QUEUE队列中只包含长度大于等于的那些串。这些串已经按从到位排好序。NONEMPTY表中列出了每一遍扫描时，哪些桶要被使用。这样可以加速计算各个桶的连接。

六．算法：

PROCEDURE Radix\_Sort2;

BEGIN

1. 建立空队列QUEUE;
2. FOR j := 0 TO DO 建立空队列Q[j];
3. FOR l := DOWNTO 1 DO

BEGIN

1. 将LENGTH[l]队列连接到QUEUE队列头；
2. WHILE QUEUE非空 DO

BEGIN

1. 设是QUEUE队列的第一个串；
2. 将从QUEUE移到队列尾；

END;

1. FOR NONEMPTY[l]中每个整数j DO

BEGIN

1. 将队列Q[j]连接到QUEUE队列尾；
2. 将Q[j]置为空队列；

END

END

END;

七．算法分析：

Radix\_Sort2算法最坏情况下时间复杂度为, 其中

。

步骤1需时间建立对偶，并需时间将它们排序。类似地，步骤2也需要时间。

现在考虑步骤3。设有个串含有第i个成份，并设含有第i个成份的串中，第i个成份有个不同值。即NONEMPTY[i]的长度为。

考虑上述算法中第3行的一个固定的l值，行的循环需时间，行的循环需时间。第4行需常数时间。行一遍需时间。整个循环共需

因为

故行的循环共需时间。第1行需常数时间，第2行需时间。从而步骤3也需时间。

八．判定两棵树同构的算法

1.定义：两棵树称为是同构的，是说：存在一个双射函数，使得对任意，当且仅当。

2.无根树的同构判定算法

算法：

(1)输入两棵树;

(2)找出的中心集合；

(3)若, 或反之，则不同构；

(4)若, 则以为根，判定是否同构；

(5)若, 则以的根，分别以为的根，判定是否同构；

(6)在第(4),(5)步中，若有一次判定出同构，则同构；否则不同构。

**§5.5 找第k个最小元**

一．问题：

输入：n个有线性序的元素组成的集合S和整数k, 。

输出：S中第k个最小元。

二．算法：

PROCEDURE SELECT (k, S);

BEGIN

1. IF |S| < 50 THEN

BEGIN

1. 用某种排序方法将S排序；
2. 返回S中第k个最小元；

END

ELSE BEGIN

1. 将S分成个子序列，每个子序列5个元素，最后一个

子序列取全部剩余元素；

1. 将每个子序列排序；
2. 设M是所有有序子序列的中间那个元素组成的集合；
3. m := SELECT();
4. 设和分别是S中小于m，等于m和大于m的

那些元素组成的集合；

1. IF THEN RETURN( SELECT(k, ) )

ELSE

10. IF THEN RETURN(m)

11. ELSE RETURN();

END

END;

三．算法分析:

设是从个元素的集合中选第个最小元所需的时间。，递归调用SELECT()至多需时间。

和中每个至多含个元素, 这是因为M中至少有个元素大于等于m, 对其中每个元素，S中有两个不同元素不小于它。因此，至多含有个元素，当时，。

类似可证。在第10行或12行的递归调用至多需时间，其它语句至多需时间。故存在常数，使得

用归纳法可证，。

故本算法需时间。

**§5.6 查找**

一．二分查找

PROCEDURE Binary\_Search (n, F, x; VAR i);

/\* 用二分查找方法在文件F中找记录x, 若x在F中，则将其序号送给i, 否则i为0。n为F的记录个数 \*/

BEGIN

l := 1; h := n; i := 0;

WHILE l h DO

BEGIN

m := (l + h) DIV 2;

IF x.key < F[m].key THEN h := m-1

ELSE IF x.key > F[m].key THEN l := m+1

ELSE BEGIN /\* x.key = F[m].key \*/

i := m;

l := h+1;

END;

END;

END;

二．算法分析

二分查找可用二叉树形象地说明。

最坏情况的时间与二叉树的深度成比例。

当, 有，因而二叉树的深度。故最坏情况下，时间复杂度为。

平均时间复杂性分析：

查找第i层(1)需要i次比较，第i层有个记录。设每个记录的查找概率相同，则。平均查找次数：

其中，l为叶结点个数，, 。

先用归纳法证明：

。

当时，，公式成立。假设公式对m成立，现在证对亦成立。

故

考虑到，故 。

于是

又考虑到

故，于是

从而平均时间复杂度为。

第六章 图的算法

**§6.1 图的遍历**

一．几个概念

1. 生成树

给定一个图，的一棵生成树是一棵树，且及。

2. 深度优先搜索：深度优先搜索的思想是，从任一顶点出发，找它的一个相邻未访问过的顶点，作为下一个要访问的顶点，再从下一个顶点出发，找它的一个相邻未访问过的顶点，作为下一个要访问的顶点，如此下去。如果当前顶点无相邻未访问的顶点，则退到上一顶点，找它的其它未访问过的相邻顶点，作为下一个要访问的顶点。

3. 宽度优先搜索：宽度优先搜索的思想是，从任一顶点出发，先逐个访问完它的所有相邻点，再从每一个相邻点出发，访问它们的所有相邻未访问的顶点。

4. 深先搜索生成树和深先搜索生成森林：按深先搜索的方法遍历图所得到的树或森林。

5. 回边和树边

例子：

图6.1

二．深度优先遍历图的算法

1. 递归算法：

输入：图和邻接表。

输出：把分成树边集和回边集。

PROCEDURE Search (v)

BEGIN

1. mark v “old”;
2. FOR each vertex w on L[v] DO

3. IF w is marked “new” THEN

BEGIN

4. add (v, w) to T;

5. Search(w);

END

END;

主程序：

BEGIN

6. T := ;

7. FOR all v in V DO mark v “new”;

8. WHILE there exists a vertex v in V marked “new” DO

9. Search(v);

END;

2.算法分析：

最坏情况下时间复杂度：，其中，n为图的顶点数，e为图的边数。

第7, 8行找所有顶点和找所有“新”顶点需时间。Search(v)过程(不含递归调用)所花的时间与跟v相邻的顶点数成正比。对每个顶点v，Search(v)只调用一次。故Search总共花的时间与成正比。故算法的时间复杂度为O(MAX(n, e))。

1. 几个性质:

引理6.1：如果(v, w)是一条回边，那么在深度优先搜索生成森林中，v是w的祖先，或相反。

证明：不失一般性，设v比w先访问。当v被访问时，w仍标记为“new”，由Search(v)访问的所有标记为“new”的顶点在生成森林中都是v的后代，并且Search(v)在w被访问前不可能结束，这因为w在中。从而有本引理的结论。

\*深度优先搜索编号：DFNUMBER[v]

4. 非递归算法:

PROCEDURE DFS (i : integer; adjlist : 图的邻接表；VAR visit :

访问标志数组)；

/\*非递归深度优先搜索遍历图，i为起始顶点标号\*/

VAR

stack : ARRAY [1max] OF integer;

top : integer; j :integer; p : 结点指针；

BEGIN

FOR j := 1 TO max DO

visit[j] := false;

WRITE(“v”, i : 1);

top := 1;

stack[top] := i; /\* 起始结点i进栈 \*/

visit[i] := true; /\* 置起始顶点已访问标志 \*/

REPEAT

p := adjlist[stack[top]]; /\* 取栈顶顶点的邻接表 \*/

WHILE (pNIL) AND visit[pvertex] DO

p := p.link; /\* 找下一个未访问过的相邻点 \*/

IF p = NIL THEN top := top1 /\*当前顶点的相邻点都已访问过\*/

ELSE BEGIN /\* 访问下一个未访问过的相邻点 \*/

i := P.vertex;

WRITE(“v”, i : 1);

visit[i] := true;

top := top+1;

stack[top] := i ;

END

UNTIL top = 0;

END;

三．宽度优先遍历图的算法

1. 算法：

PROCEDURE BFS (i : integer; adjlist : 图的邻接表；VAR visit :

访问标志数组)；

/\* 非递归宽度优先搜索遍历图，i : 起始顶点标号 \*/

VAR

queue : ARRAY [1max] OF integer;

j, hp, tp : integer;

p : 顶点指针；

BEGIN

FOR j := 1 TO max DO

visit[j] := false;

hp := 1; tp := 1;l

WRITE(“v”, i : 1); /\* 输出起始顶点 \*/

visit[i] := true; /\* 置起始顶点已访问标志 \*/

queue[hp] := i ; /\* 起始顶点进队列 \*/

REPEAT

p := adjlist[queue[hp]]; /\* 取队列头顶点的邻接表 \*/

WHILE p NIL DO

BEGIN /\* 当该顶点有相邻顶点时做 \*/

i := p.vertex; /\* 取该相邻点的编号 \*/

IF NOT visit[i] THEN /\* 若未访问过 \*/

BEGIN

WRITE(“v”, i : 1);

visit[i] := true; /\* 置已访问标志 \*/

tp := tp+1;

queue[tp] := i ; /\* 顶点i进队列 \*/

END;

p := p.link;

END;

hp := hp+1; /\* 队列头元素出队列 \*/

UNTIL hp > tp;

END;

2. 算法分析

最坏情况下时间复杂度：。

**§6.2 图的2-连通性**

一．几个概念：

1. 割点：一个顶点a称为G的割点，是说：存在G的两个顶点v和w，a, v, w互不相同，并且G中任意一条从v到w的路径都包含顶点a。换句话说，从G中删除a及与a相关联的边，G将被分割成两个或更多的连通分支。

2. 2-连通性：图G称为是2连通的，是说：对G中任意3个不同的顶点v, w, a，存在一条从v到w的路径不包含a。无向连通图G是2连通的，当且仅当G中不含割点。

3. 2-连通分支：

\*即G中的极大子图，该子图是2-连通的。

定义图G的边集E的等价关系R。当且仅当或存在G中的圈包含。

等价关系R将E划分成等价类，任意两条边属于同一等价类当且仅当这两条边同在某一圈上。对令为中边的端点的集合，则称为G的2-连通分支。

图6.2

二．2-连通分支和割点的性质

引理6.2：对为连通无向图的2连通分支。那么

1. 对每个，是2连通的；
2. 对任意至多含一个顶点；
3. a是G的割点当且仅当对某两个。

证明：1.假设中存在3个不同顶点v, w, a，使得中所有从v到w的路径都经过a，则不是中的边。因此存在中的边和，并且存在中的圈包含这两条边。由2-连通分支的定义，这个圈上的所有边和顶点都分别属于。故在中，v到w有两条内部不相交的路径，而a不可能同时包含在这两条路径中，矛盾。

2. 假设中有两个不同的顶点v和w。那么中存在圈包含v和w，中存在圈也包含v和w。且的边不相交。从而我们可以用的边构成一个圈C，C中既有的边，又有的边。因而中至少有一条边与中一条边等价，这就证明了不是两个等价类，矛盾。

3. 设a是G的一个割点。那么存在G中两个顶点v和w，v, w, a互不相同且每一条从v到w的路径包含a。因为G是连通的，故至少存在一条这样的路径。设是这条路径上两条与a关联的边。如果存在圈包含这两条边，则存在从v到w的路径不包含a。故和属于不同的2连通分支。故a属于它们的顶点集的交。

反过来，如果，那么存在边和分别属于。因为这两边不包含在任何圈中，故从x到y的每一条路包含a。故a是割点。

引理6.3：设是连通无向图，设是G的深度优先搜索生成树。顶点a是G的割点当且仅当或者

1. a是根结点且a有多于一个儿子；或者
2. a不是根，并且对于a的某个儿子s，不存在回边从s的任一后代(包括s本身)到a的任一真祖先。

证明：易证，一个根结点是割点当且仅当它有多于一个儿子。

假设条件2为真。设f是a的父亲。由引理6.1，每一条回边从一个顶点到它的祖先。因此，任意回边从s的后代v到v的祖先。由本引理的假设，这些回边不能到达a的真祖先。因此它们到达a或s的后代。从而每一条从s到f的路径包含a，即a是割点。

假设a是割点，但不是根。设x和y是不同于a的两个不同顶点，并且G中从x到y的每条路径包含a。则x和y中至少有一个(例如x)是a在S中的真后代。假若不然，S中存在从x到y的路径，该路径不包含a。设s是a的儿子使得x是s的后代(可能)。这时，或者不存在从s的后代r到a的真祖先w的回边，从而条件2成立；或者存在一条这样的回边。在后一个情形下，我们考虑两个情形：

情形1：设y不是a的后代。那么存在从x到v到w再到y的路径，该路径不含a，矛盾。

情形2：设y是a的后代。则y不是s的后代，否则存在从x到y的路径不含a。设s’是a的儿子，使得y是s’的后代。那么或者不存在s’的后代v’到a的真祖先w’的回边，从而条件2成立；或者存在这样一条回边。在后一种情形下，存在一条路径从x到v到w到w’到v’再到y，该路径不包含a,矛盾。从而条件2为真。 证毕。

三．几个定义：

设连通无向图的深度优先搜索生成树的树边集为T，回边集为B,并设V中顶点以深度优先搜索的序号命名。对任意，定义

LOW[v]=MIN({v}{w | 存在回边，使得x是v的后代，并且w是v的祖先}) (6.1)

\*由于深度优先搜索生成树中顶点标号是按前序遍历的顺序标号的，如果x是v的后代，并且是回边，满足，则w是v的真祖先。

由引理6.3，如果顶点v不是根，那么v是割点当且仅当v有一个儿子s，使得LOW[s]v。

LOW[v]可以通过确定以下顶点w的最小值来计算：

1. ，或
2. 且s是v的一个儿子，或
3. 是B中的一条回边。

(6.1)式等价于

LOW[v]=MIN({v}{LOW[s] | s是v的一个儿子}{w | (v, w)B}) (6.2)

四．算法

输入：一个连通无向图

输出：G的每个2-连通分支的边集

方法：

1. 将集合T初始化为，COUNT初值为1。将V中每一个顶点标记为“new”。选择V中任一顶点，调用过程SEARCHB()，建立深度优先搜索生成树，并对V中每个v计算LOW[v]。
2. 当SEARCHB的第5行遇到顶点w，如果边不在栈STACK中，则将其放入STACK。当在第10行发现一对顶点满足：w是v的儿子且LOW[w]后，把STACK从栈顶到(v, w)的所有边弹出来。这些边构成G的一个2-连通分支。

PROCEDURE SEARCHB (v);

BEGIN

1. Mark v “old”;
2. DFNUMBER[v] := COUNT;
3. COUNT := COUNT+1;
4. LOW[v] := DFNUMBER[v];
5. FOR each vertex w on L[v] DO
6. IF w is marked “new” THEN

BEGIN

1. add (v, w) to T;
2. FATHER[w] := v;
3. SEARCHB(w);
4. IF LOW[w]DFNUMBER[v] THEN

a biconnected component has been found;

1. LOW[v] := MIN(LOW[v], LOW[w]);

END

1. ELSE IF w is not FATHER[v] THEN

LOW[v] := MIN(LOW[v], DFNUMBER[w]);

END;

五．算法分析

步骤1的时间为O(e)，按6.1节的Search过程的分析，可得本结论。步骤2考察每一条边一次，将其放入栈中，然后弹出来，故步骤2的时间为O(e)。总的时间为：O(e).

六．算法的正确性

定理6.4：上述算法正确地找出图G的所有2连通分支。

证明：引理6.3保证了上述算法正确地找出所有割点。即使根不是割点，我们也把它当作割点处理，以便找出包含根的2连通分支。

以下证明，如果，那么当SEARCHB(w)结束时，栈STACK中边及以上的所有边恰好是包含边的2连通分支。我们对G的2连通分支个数b归纳证明。当b = 1时，因为是树的根，是关联的唯一树边。当SEARCHB(w)完成时，G的所有边在STACK中，故得证。

现在设归纳假设对所有具有b个2连通分支的图成立，并设G具有b + 1个2连通分支。设SEARCHB(w)是第一次满足的对SEARCHB的调用，其中是树边。因为STACK中没有弹出任何边，STACK中及其以上的边恰好是与及其后代关联的所有边。易证，这些边恰好是包含的那个2连通分支的边。从栈STACK中移去这些边后，本算法的工作恰如处理从G中删去包含的那个2连通分支所得到的图时所做的工作。而恰好包含b个2连通分支，由归纳假设，本算法正确地找出的所有2连通分支，从而正确地找出G的所有2连通分支。