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。

作业3：

1. 解以下递归方程，其中, 对，满足以下式子：

给出的和估计。

1. 给出以下递归方程的的和估计：

。

1. 对n个Merge-Find操作问题设计一个的算法。使用

这样一棵树作数据结构，它的所有叶子与树根的距离为2，限制树根的度为1到之间，限制树根的每个儿子的度为1到之间。