第十一章 高级线性表

任课教员：张铭、赵海燕、冯梅萍、王腾蛟
http://db.pku.edu.cn/mzhang/DS/
北京大学信息科学与技术学院

主要内容

- 11.1 多维数组
- 11.2 广义表
- 11.3 存储管理技术

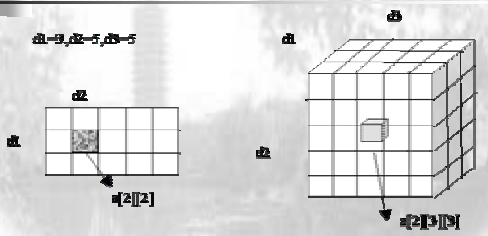
11.1 多维数组

- 数组（Array）是数量和元素类型固定的有序序列
  - 静态数组必须在定义它的时候指定其大小和类型
  - 动态数组可以在程序运行才分配内存空间
- 多维数组（Multi-array）是向量的扩充
  - 向量的向量就组成了多维数组
  - 可以表示为：

    ELEM A[c_1..d_1][c_2..d_2]...[c_n..d_n]
  - c_i和d_i是各维下标的下界和上界。所以其元素个数为：

$$\prod_{i=1}^{n}(d_i - c_i + 1)$$

数组的空间结构



左边是二维数组的空间结构，右边是三维数组的空间结构，d1[1..3],d2[1..5],d3[1..5]分别为3个维。

数组的存储

- 内存是一维的，所以数组的存储也只能是一维的
  - 以行为主序（也称为“行优先”）
  - 以列为主序（也称为“列优先”）
- 一个3 × 3的数组X的行优先表示：

$$X=\begin{vmatrix}1&2&3\\4&5&6\\7&8&9\end{vmatrix}$$

  - 内存中的存放是：1，2，3，4，5，6，7，8，9

数组的存储（续）

- 一个二维m × n数组中元素X[i][j]（第i行第j列元素）的内存地址可以这样来计算：
  - X[0][0]（数组首地址）+（n×i+j） × 元素的长度（如C++中int型为4字节）
- 例如，我们已知一个数组的A[0][0]元素在内存的644的位置，假设元素的长度为8，那么我们就可以求得其他任意元素A[x][y]的位置，为644+len×(n×x + y)。
  - 例如，n = m = 3，由上面公式得到A[2][3]元素的地址：644 + 8×（3×2+2）= 708

- Pascal语言的存储实现是按行优先处理的，先排最右的下标，从右向左，最后最左的下标。
- 例如对于三维数组a[1..k,1..m,1..n]的元素a_{xyz}可以如下排列：

Pascal语言的行优先存储

$$
\begin{array}{llll}
a_{111} & a_{112} & a_{113} & \cdots\ a_{11n} \\
a_{121} & a_{122} & a_{123} & \cdots\ a_{12n} \\
\multicolumn{4}{c}{\cdots\cdots\cdots\cdots} \\
a_{1m1} & a_{1m2} & a_{1m3} & \cdots\ a_{1mn} \\
a_{211} & a_{212} & a_{213} & \cdots\ a_{21n} \\
a_{221} & a_{222} & a_{223} & \cdots\ a_{22n} \\
\multicolumn{4}{c}{\cdots\cdots\cdots\cdots} \\
a_{2m1} & a_{2m2} & a_{2m3} & \cdots\ a_{2mn} \\
\multicolumn{4}{c}{\vdots} \\
a_{k11} & a_{k12} & a_{k13} & \cdots\ a_{k1n} \\
a_{k21} & a_{k22} & a_{k23} & \cdots\ a_{k2n} \\
\multicolumn{4}{c}{\cdots\cdots\cdots\cdots} \\
a_{km1} & a_{km2} & a_{km3} & \cdots\ a_{kmn}
\end{array}
$$

- FORTRAN语言采用列优先存储。先排最左的下标，从左向右，最后最右的下标。
- 例如对于三维数组a[1..k, 1..m, 1..n]的元素a_{xyz}可以如下排列：

FORTRAN语言的列优先存储

$$
\begin{array}{llll}
a_{111} & a_{211} & a_{311} & \cdots\ a_{k11} \\
a_{121} & a_{221} & a_{321} & \cdots\ a_{k21} \\
\multicolumn{4}{c}{\cdots\cdots\cdots\cdots} \\
a_{1m1} & a_{2m1} & a_{3m1} & \cdots\ a_{km1} \\
a_{112} & a_{212} & a_{312} & \cdots\ a_{k12} \\
a_{122} & a_{222} & a_{322} & \cdots\ a_{k22} \\
\multicolumn{4}{c}{\cdots\cdots\cdots\cdots} \\
a_{1m2} & a_{2m2} & a_{3m2} & \cdots\ a_{km2} \\
\multicolumn{4}{c}{\vdots} \\
a_{11n} & a_{21n} & a_{31n} & \cdots\ a_{k1n} \\
a_{12n} & a_{22n} & a_{32n} & \cdots\ a_{k2n} \\
\multicolumn{4}{c}{\cdots\cdots\cdots\cdots} \\
a_{1mn} & a_{2mn} & a_{3mn} & \cdots\ a_{kmn}
\end{array}
$$

行优先存储公式

- 设数组元素占d个存储单元，3维矩阵行优先存储公式为：

$$loc(A[j_1, j_2, j_3]) = loc(A[1,1,1])$$

$$+ d \cdot [(j_1 - 1) \cdot m \cdot n + (j_2 - 1) \cdot n + (j_3 - 1)]$$

n维矩阵行优先存储公式为：

$$
\begin{aligned}
& loc(A[j_1, j_2, \ldots, j_n]) = loc(A[c_1, c_2, \ldots, c_n]) \\
& + d \cdot [(j_1 - c_1)(d_2 - c_2 + 1)\ldots(d_n - c_n + 1) \\
& + (j_2 - c_2)(d_3 - c_3 + 1)\ldots(d_n - c_n + 1) \\
& + \ldots + (j_{n-1} - c_{n-1})(d_n - c_n + 1) + (j_n - c_n)] \\
& = loc(A[c_1, c_2, \ldots, c_n]) \\
& + d \cdot [\sum_{i=1}^{n-1}(j_i - c_i)\prod_{k=i+1}^{n}(d_k - c_k + 1) + (j_n - c_n)] \\
& = loc(A[c_1, c_2, \ldots, c_n]) + d \cdot [\sum_{i=1}^{n}(j_i - c_i)\prod_{k=i+1}^{n}(d_k - c_k + 1)]
\end{aligned}
$$

- C++多维数组ELEM A[d_1][d_2]...[d_n];

$$loc(A[j_1, j_2, \ldots, j_n]) = loc(A[0,0,\ldots,0])$$
$$+d \cdot [j_1 \cdot d_2 \cdot \ldots \cdot d_n + j_2 \cdot d_3 \cdot \ldots \cdot d_n$$
$$+\ldots + j_{n-1} \cdot d_n + j_n]$$
$$= loc(A[0,0,\ldots,0]) + d \cdot [\sum_{i=1}^{n-1} j_i \prod_{k=i+1}^{n} d_k + j_n]$$

数组的声明

- 在编译的时候如果已经知道数组每一维的大小：

  声明一个10 × 10的整型数组：int num[10][10];
- 只知道数组一个维的大小，那么也可以动态地创建一个二维数组。例如我们只要一个组有10个整数，但是不知道有多少个组：int (*num)[10];
  - 最后在程序运行的时候，可以计算出或者由用户指定它的第一个维数是n：
  - num= new int[n][10];

数组的声明:动态的声明

```
int **X;
int row=3;
int col=3;
try{
  //创建行指针，即指向整型的指针
  X= new int*[row];
  //然后再为每一行分配地址
  for(int i=0; i<row;i++)
  {
    X[i]=new int[col];
   }
}
catch(xalloc){…}
```


数组的声明:动态的声明

- 这里要注意的是，内存被分配出去也必须加以回收，否则会造成内存泄漏（memory leak）
- for(int i=0;i<row;i++)
  delete[] X[i];
  delete []X;

对于3维或者更高维的数组，过程是类似的

用数组表示特殊矩阵

- 二维数组可以被看作是矩阵，所以它也经常被用来表示矩阵
- 三角矩阵可以被分为上三角矩阵和下三角矩阵，它是指n阶矩阵中的下(上)三角的元素都是0或者一个常数
  - 在上三角矩阵中，当数组的下标i>j时，数组元素a[i][j]=c；
  - 而在下三角矩阵中，当下标i<j时，a[i][j]=c。
- 假设以一维数组list[0..(n^2+n)/2-1]作为n阶下三角矩阵A的存储结构
  - 原来矩阵中的元素$a_{i,j}$与线性表相应元素的对应位置为list[(i^2+i)/2 + j] (i>=j)

下三角矩阵图例

$$\begin{pmatrix} 0 & & & & & \\ 0 & 0 & & & & \\ 7 & 5 & 0 & & & \\ 0 & 0 & 1 & 0 & & \\ 9 & 0 & 0 & 1 & 8 & \\ 0 & 6 & 2 & 2 & 0 & 7 \end{pmatrix}$$

用数组表示特殊矩阵（续）

- 对称矩阵指的是一个n阶矩阵，它的元素满足性质 $a_{i,j}=a_{j,i}$，0<=（i, j）<n。在用数组存储的时候，元素a[i][j]=a[j][i]

$$\begin{pmatrix} 0 & & & & & \\ 0 & 0 & & & & \\ 7 & 5 & 0 & & & \\ 0 & 0 & 1 & 0 & & \\ 9 & 0 & 0 & 1 & 8 & \\ 0 & 6 & 2 & 2 & 0 & 7 \end{pmatrix}$$

- 为了节省空间，存储其下三角的值，而对角线之上的值通过对称关系映射过去。
- 以一维数组sa[0..n(n+1)/2]作为n阶对称矩阵A的存储结构，则sa[k]和矩阵元$a_{i,j}$之间存在着一一对应的关系 ：

$$k=\begin{cases}\dfrac{j(j+1)}{2}+i, & \text{当} i<j \\ \dfrac{i(i+1)}{2}+j, & \text{当} i\geq j\end{cases}$$

用数组表示特殊矩阵（续）

- 对角矩阵是指所有的非零元素都集中在主对角线及以它为中心的其他对角线上。如果|i-j|>1，那么数组元素a[i][j]=0。
  - 下面是一个3对角矩阵：

$$\begin{pmatrix} a_{0,0} & a_{0,1} & \cdots\cdots & 0 \\ a_{1,0} & a_{1,1} & a_{1,2} & \cdots\cdots \\ & & \ddots & \ddots & a_{n-1,n-2} \\ 0 & & a_{n-1,n-2} & a_{n-1,n-1} \end{pmatrix}$$

稀疏矩阵

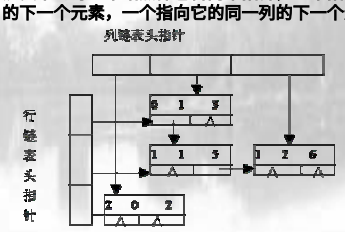
- 稀疏矩阵中的非零元素非常少，而且分布也不规律
- 稀疏因子
  - 用来描述稀疏矩阵的非零元素情况，它定义为在m ×n的矩阵中，有t个非零元素，则稀疏因子为：

$$\delta=\frac{t}{m\times n}$$

  - 通常当这个值小于0.05时，可以认为是稀疏矩阵
- 一般使用三元组(i，j，a_{ij})来顺序存储稀疏矩阵，其中i是该元素的行号，j是该元素的列号，a_{ij}是该元素的值

稀疏矩阵的十字链表

- 十字链表有两组链表组成
  - 行和列的指针序列
  - 链表中的每一个结点都包含两个指针，一个指向它的同一行的下一个元素，一个指向它的同一列的下一个元素



十字链表的ADT

```
template<class T>
class OLNode
{
private:
  int row,col;//矩阵的行和列
  T element;//矩阵中存储的数据
  OLNode<T>* right,*down;//指向下一个结点的指针
  public:
      OLNode(){right=NULL;down=NULL;};
};
```

十字链表的建立

- 建立矩阵的算法如下：
  - 首先为行头结点和列头结点申请空间，大小分别为矩阵的行列数
  - 将三元组根据情况分别加入到链表中
    - 如果三元组中的行列号错误，则退出，否则继续
    - 先处理行链表的问题
      - 如果该行头结点为空，则建立一个新的头结点，内容为该三元组
      - 如果不为空则从头结点开始查找，找到该三元祖的正确位置如果该位置已经存在数据，则修改之，否则生成相应的结点插入进去
    - 类似地处理列链表头

矩阵乘法

$$\begin{bmatrix} 3 & 0 & 0 & 5 \\ 0 & -1 & 0 & 0 \\ 2 & 0 & 0 & 0 \end{bmatrix} \times \begin{bmatrix} 0 & 2 \\ 1 & 0 \\ -2 & 4 \\ 0 & 0 \end{bmatrix}$$

$$= \begin{bmatrix} 0 & 6 \\ -1 & 0 \\ 0 & 4 \end{bmatrix}$$

经典矩阵乘法

- A[c1..d1][c3..d3]，B[c3..d3][c2..d2]，C[c1..d1][c2..d2]。
- $C=A\times B\ (C_{ij}=\sum_{k=c3}^{d3} A_{ik}\cdot B_{kj})$
-

```
for (i=c1; i<=d1; i++)
  for (j=c2; j<=d2; j++) {
    sum = 0;
    for (k=c3; k<=d3; k++)
      sum = sum + A[i,k]*B[k,j];
    C[i, j] = sum;
  }
```


- p=d1-c1+1，m=d3-c3+1，n=d2-c2+1；
- A为p×m的矩阵，B为m×n的矩阵，乘得的结果C为p×n的矩阵
- 经典矩阵乘法所需要的时间代价为O(p×m×n)

稀疏矩阵乘法

```
template <class T>
SMatrix<int>*SMatrix<T>::MatrixMutil(SMatrix<int>*left,S
   Matrix<int>*right)
{
   if(left->GetColnum()!=right->GetRownum())
        return NULL;//行列不匹配不能相乘

   int I=0; //第一个矩阵的行数
   int J=0; //第二个矩阵的列数
   SMatrix<T> *ResultMatrix=new SMatrix<T>();//结果矩阵
   ResultMatrix->MallocMem(left->GetRownum(),right-
    >GetColnum());//为结果矩阵分配空间
   for(I=1;I<=left->GetRownum();I++)
   {//开始相乘
        OLNode<T>* RowNext=ResultMatrix->rowhead[I];
```



```
for(J=1;J<=right->GetColnum();J++)
{//扫描所有的列
        OLNode<T>* ColNext=ResultMatrix->colhead[J];
        int result=0;
        OLNode<int> * rows=left->rowhead[I];
        OLNode<int>* cols=right->colhead[J];
        if((rows==NULL)||(cols==NULL))
                break;//新行没有非零元素

        while((rows!=NULL)&&(cols!=NULL))
        {//所有行列都未非零的元素相乘
                if(rows->col<cols->row)
                {
                        rows=rows->right;
                }
                else
                        if(rows->col>cols->row)
                        {
                                cols=cols->down;
                        }
```

```
                else
                {//都有元素可以相乘
                    result=result+cols->element*rows->element;
                    cols=cols->down;
                    rows=rows->right;
                }
            }
                if(result==0) continue;
                //插入到结果矩阵中
                OLNode<T>* temp=new OLNode<T>();
                temp->row=I;
                temp->col=J;
                temp->element=result;
                if(RowNext==NULL)//加入行向量中
                {//每行第一个元素
                    ResultMatrix->rowhead[I]=temp;
                    RowNext=ResultMatrix->rowhead[I];
                }
                else
                {//加入一个新的元素到下一个位置
                        RowNext->right=temp;
                        RowNext=RowNext->right;
                }
```



```
                if(ColNext==NULL)//加入到列向量
                {//列结点第一次使用，加入新的结点
                        ResultMatrix->colhead[J]=temp;
                        ColNext=ResultMatrix->colhead[J];
                }
                else
                {//调转到合适的位置，插入
                        while(ColNext->down!=NULL)
                                ColNext=ColNext->down;
                        ColNext->down=temp;
                        ColNext=ColNext->down;
                }
        }//完成放入ResultMatrix

    }
    ////////////////////乘法做完，返回新的矩阵
    return ResultMatrix;
}
```


稀疏矩阵乘法时间代价

- 若矩阵A中行向量的非零元素个数最多为t_a
- 矩阵B中列向量的非零元素个数最多为t_b
- 矩阵C中列向量的非零元素个数最多为t_c
- 假设C矩阵中非0元素的个数总和为N_c

- 每计算一个 c_{ij}的时间
  - 主要用于顺着行I和列J寻找的过程
  - 其循环次数最多为t_a+t_b
  - 每次循环所花时间是一个常量，记为k
  - 计算一个c_{ij}的时间为$k\times(t_a+t_b)$
  - 计算全部c_{ij}的时向为$k\times(t_a+t_b)\times p\times n$。

- 生成乘积矩阵的时间
  - 计算出来的结果在行向量里面插入的时间是常数
  - 在列向量上插入需要每次定位到合适的位置， 所花时间为$O(t_c)$，插入操作的总时间为$O(N_c\times t_c)$
- 因此稀疏矩阵乘法的总执行时间上界为$O((t_a+t_b)\times p\times n)+O(N_c\times t_c)$
- 如果修改此算法
  - 保留乘积C矩阵的当前列指针向量位置，指向已插入到C中各列最新的非零元素
  - 可使每次插入的时间为一常数
  - 总执行时间降低为$O((t_a+t_b)\times p\times n)$

11.2 广义表

- 回顾线性表
  - 由n（n≥0）个数据元素组成的有限有序序列
  - 线性表的每个元素都具有相同的数据类型，通常为同一某种类型的数据记录
- 如果一个线性表中还包括一个或者多个子表，那就称之为广义表（Generalized Lists，也称Multi-list）一般记作：
  - $L=(x_0, x_1, \ldots, x_i, \ldots, x_{n-1})$

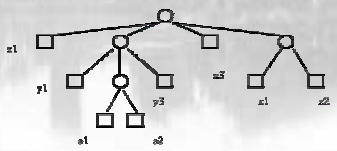
$L=(x_0, x_1, \ldots, x_i, \ldots, x_{n-1})$

- L是广义表的名称
- n为长度
- 每个x_i（$0 \le i \le n-1$）是L的成员
  - 可以是单个元素，即原子（也称“单元素”，atom）
  - 也可以是一个广义表，即子表（sublist）
- 广义表的深度：表中元素都化解为原子后的括号层数

广义表的各种类型

- 纯表（pure list）
  - 从根结点到任何叶结点只有一条路径
  - 也就是说任何一个元素（原子、子表）只能在广义表中出现一次

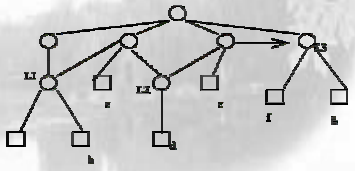
(x1, (y1 ,(z1 ,z2), y3), x3 ,(z1 ,z2))



广义表的各种类型（续）

- 可重入表 (reentrant list)
  - 图示对应于一个DAG
  - 其元素(包括原子和子表)可能会在表中多次出现
    - 但不会出现回路
- 对子表和原子标号

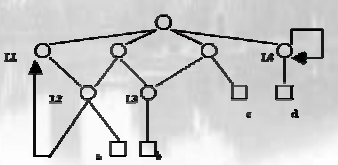
(((a, b)), ((a, b), c, d), (d, e, f, g) , (f, g))

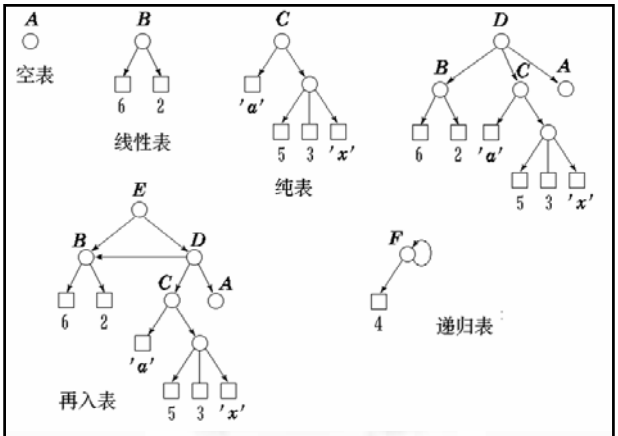


广义表的各种类型（续）

- 循环表(cyclic list ，递归表)的图示对应于任何有向图
  - 有向图中可能包含回路
  - 循环表的深度为无穷大

(L1:(L2:(L1, a)), (L2, L3:(b)), (L3, c), L4:(d, L4))





- 图⊇递归表⊇再入表⊇纯表(树)⊇线性表
  - 广义表是线性与树型结构的推广

广义表的存储

- 使用数组来存储
  - 存储括号

(	x1	(	y1	(	a1	a2	)	y3	)	x3	(	z1	z2	)	)

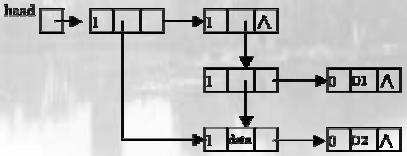
- 可以使用链表结构存储

广义表存储ADT

```
template <class T>
class GenListNode
{
  public:
        int type;   //表示该结点是ATOM或者SubLIST
        T element; //如果是ATOM，则存储它的值
        //如果是LIST ,则指向它的元素的首结点
        GenListNode<T> *child;
        GenListNode<T> *next;//指向下一个结点
        ...  // 其他函数
 };
```

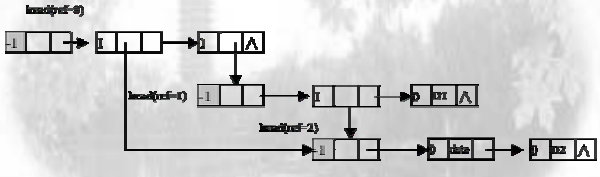

广义表存储ADT（续）

- 不带表头的广义表链
  - 在删除结点的时候会出现问题
  - 删除结点data就必须进行链调整

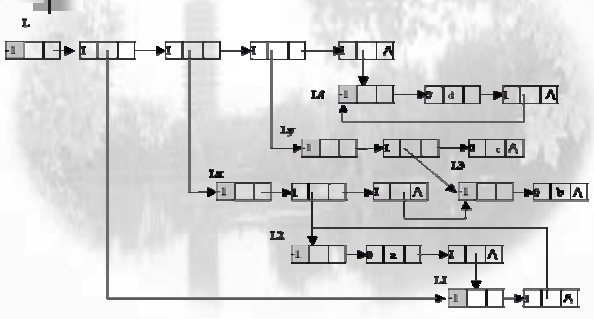


广义表存储ADT（续）

- 增加一个头指针来标志，简化删除，插入操作



- 此外，在访问循环链的时候会出现循环访问，所以需要一个标志位来记录该结点是否已经被访问过
  - 图的因素




```
//改进的广义表结点类型
template <class T>
class GenListNode
{
    public:
          int type;   //表示该结点是ATOM or LIST
          struct {
             int ref;   //如果是表头结点则，存储该结点被引用次数
             char* Name;  // 表头名称
           int mark;   //本子表是否被访问过
          } headNode;
          GenListNode<T> *child;//如果是LIST ,则指向子表
          T element; //如果是ATOM，则存储它的值

          GenListNode<T> *next;//指向下一个结点
          void GenListTraversal ();//周游该结点的子孙
          void GenListTraversalHelp(GenListNode<T> *node);
          ......
};
```

```
//GenList是一个原子元素类型为T的广义表
//如果要实现能够存储多种数据类型的广义
//表，只需要嵌套的使用它
template <class T>
class GenList
{
  private:
      GenListNode<T> *head;//头结点,存储头信息
      GenListNode<T> *current;//当前指针的位置

  public:
      GenList(char *Name);
      ~GenList();
      void Insert(T element);//在尾部加入一个元素结点
      void Insert(GenList<T> *gl);//在尾部插入一个子表
       GenListNode<T> *GetHead();//取头结点
      GenListNode<T> *GetNext();//取当前结点的下一个
      GenListNode<T> *GetPrev();//取当前结点的前一个
      void MoveFirst();//当前指针指向head
      int Remove();//删除当前结点
      void ViewList();//周游广义表
};
```

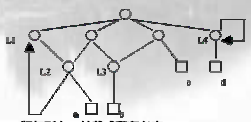

广义表的周游算法

- 广义表周游的时候应该注意几个问题：
  - 相当于深度优先周游
  - 访问的时候首先进入一个子表的头结点，设置mark标记
  - 按照本层子结点顺序，访问广义表
    - 如果是子表结点，则准备递归地访问此子表表头结点
    - 如果是原子，则直接访问
  - 避免进入循环链中无法跳出
    - mark用来防止循环访问而设置的访问位
    - 实际上，表头结点才需要mark
- 广义表访问结束的时候
  - 应该将mark设置为未访问
  - 以便如果其他地方也引用了该链可以正常的访问到


```
GenList<char*> *List=new GenList<char*>("List");
GenList<char*> *List1=new GenList<char*>("L1");
GenList<char*> *List2=new GenList<char*>("L2");
GenList<char*> *List3=new GenList<char*>("L3");
GenList<char*> *List4=new GenList<char*>("L4");
GenList<char*> *Listx=new GenList<char*>("");
GenList<char*> *Listy=new GenList<char*>("");
List3->Insert("b");
List4->Insert("d");      List4->Insert(List4);
Listy->Insert(List3);    Listy->Insert("c");
List2->Insert("a");      List2->Insert(List1);
List1->Insert(List2);
Listx->Insert(List2);    Listx->Insert(List3);
List->Insert(List1);     List->Insert(Listx);
List->Insert(Listy);     List->Insert(List4);
```

(L1:(L2:(L1, a)), (L2, L3:(b)), (L3, c), L4:(d,L4))

```
List->ViewList();
```




```
template <class T>
void GenListNode<T>::GenListTraversalHelp(GenListNode<T>
   *node)  {
   GenListNode<T> *p;
   node->headNode.mark=VISITED;
   cout << "(";
   for (p = node->next; p!=NULL; p=p->next) {
        //进入一个子表结点，准备递归访问它的表头结点
        if ((p->type==LIST)&&(p->child!=NULL)) {
           cout << p->child->headNode.Name;
           if (p->child->headNode.mark == UNVISITED)  {
                if (p->child->headNode.Name[0]!='\0')
                     cout <<":";
                GenListTraversalHelp(p->child);
           }
        }
        else if (p->type==ATOM)  cout<<p->element;
        if ((p->next!=NULL)) // &&(p->next->type!=HEAD))
           cout << ", ";
   }
   cout << ")";
```



```
template <class T>
void GenList<T>::ViewList()
{
    MoveToFirst();
   current->GenListTraversal ();
}

template <class T>
void GenListNode<T>::GenListTraversal ()
{
   GenListTraversalHelp(this);
}
```


11.3 存储管理技术

- 在C++这样的高级程序设计语言中，程序员经常会进行new和delete操作，即动态内存分配。在实际的操作系统中，内存管理本质上是利用链表和广义表来实现的。
- 本小节将首先介绍可利用空间表的概念，并简单介绍可利用空间表中所有结点长度都相等的情况。
- 随后将介绍在可利用空间表中处理变长结点的方法

分配与回收

- 内存管理最基本的问题是存储的分配与回收
- 分配存储空间，回收被“释放”的存储空间。在分配和回收过程中，需要解决碎片问题，这就是存储的压缩
- 可能由于程序员忘记delete已经不再使用的指针等等，而产生了许多无用单元(garbage)，需要对这样的无用单元进行有效的收集，而且收集完往往还需要再进行压缩。

存储空间溢出的管理

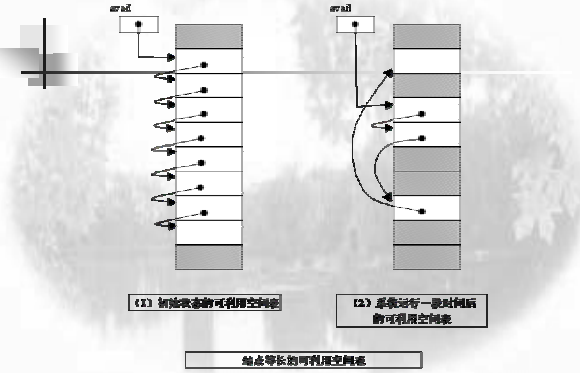
- 由于主存容量的限制，所以无论采用哪种存储管理技巧，总难免产生内存溢出。
- 溢出发生后，可以借助外存的帮助，把内存中某些结点（或由这些结点组成的结构）撤离到外存上去，并且提供一定的手段在必要时将这些结点再取回内存。
- 为了减少内外存数据交换次数，送到外存上去的内容应该选择最近不使用的那些结点

可利用空间表

- 为了进行动态存储分配，可以把存储器看成一组变长块数组，其中一些块是空闲的，一些块是已分配的，空闲块链接到一起，形成一个可利用空间表(freelist)。
- 所谓可利用空间是指存储区中当前还没有使用的空间
- 对于存储请求，要在可利用空间表中找到足够大的块。如果找不到，那么存储管理器就要求助于失败策略。

长度固定的存储分配

- 把可利用空间表组织成链栈(或链式队列)的形式。
- 在系统运行初期将整个可利用空间划分成固定大小的数据块，而且利用指针字段把这些数据块链接起来，并使用一个指针指向首结点，这样就形成了一个单链表即这个可利用空间表。
- 以后每执行一次new p操作就从可利用空间中取走一个数据块，并用p指向该数据块；每执行一次delete p操作就把p指向的数据块插入到可利用空间表的链表中。



可利用空间表的单链表定义和函数重载

```
template <class Elem> class LinkNode{
  private:
      static LinkNode *avail;       //可利用空间表头指针
  public:
      Elem value;   //结点值
      LinkNode * next;      //指向下一结点的指针
      LinkNode(const Elem & val, LinkNode * p);
      LinkNode(LinkNode * p = NULL);//构造函数
      void * operator new(size_t);//重载new运算符
      void operator delete(void * p);       // 重 载 delete 运
  算符
};
```

```
//重载new运算符实现
template <class Elem>
void * LinkNode<Elem>::operator new(size_t){
   if(avail == NULL)//可利用空间表为空
        return ::new LinkNode;      //利用系统的new分配空间
   LinkNode<Elem> * temp = avail;         //否则从可利用空间表中取走一个结点
   avail = avail->next;
   return temp;
}
```



```
//重载delete运算符实现
template <class Elem>
void LinkNode<Elem>::operator delete(void * p){
   ((LinkNode<Elem> *) p)->next = avail;
   avail = (LinkNode<Elem> *)p;
}
```


- 这种可利用空间表实际上是一个用单链表实现的栈。new代表栈的删除操作，如果avail为空指针，代表已没有可利用空间。delete代表栈的插入操作。
- 如果程序员需要直接引用系统的new和delete操作符，需要强制用“::new p”和“::delete p”。这种强制在整个程序运行完毕时是十分必要的，可以把avail所占用的空间都交还给系统（真正释放空间）。

各种类型和长度的可利用空间表

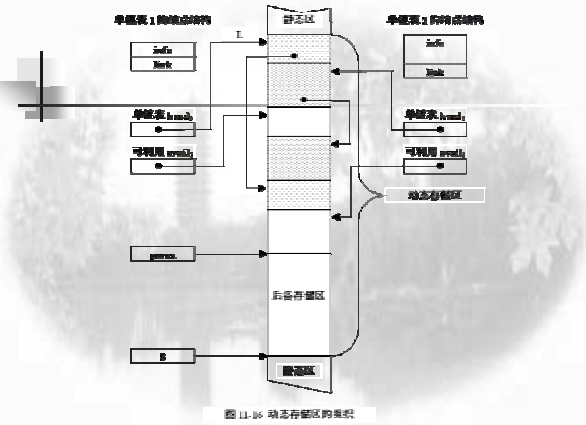
- 有三种很直观的解决方案：
- (1) 建立起多个可利用空间表，每个链表可以为某一种长度的变量分配存储空间。
- (2) 统一按照较长的结点组织可利用空间表，把变长结点按同样长度进行分配。这在结点长度差别不大时还可以采用，但是在长度差别很大时，就可能造成不可容忍的存储空间浪费。
- (3) 多个可利用空间表共享同一个存储空间，事先估计出每个链表中最多可以有多少个结点，并把这些结点都链接起来。但这造成了在空间和时间上的巨大浪费。这样的处理没有解决共享问题，代价很高，管理也不方便。

动态分配

- 不对每个可利用空间表进行预分配，而是随着系统运行而动态分配。
- 假设有一片从地址L开始的动态存储区域，上界地址为S。这片存储区域由n个链表所共享，每个链表的结点类型都不同。显然，需要为这n个链表建立n个可利用空间表。
- 系统刚开始运行，所有的可利用空间表的头指针avail都赋为空值。
- 在系统运行过程中，每当链表的结点被删除时，把被删除结点推入到对应的可利用空间表中储备起来。

动态分配（续）

- 每当需要向某链表中动态插入结点时，如果对应的可利用空间表非空，则从可利用空间表中删除一个结点的空间给它；如果对应的可利用空间表为空，则从后备存储区中去取一块存储空间。
- 我们用一个指针pmax来指向动态存储区的后备存储区的起始地址，随着后备存储区的不断消耗，pmax值不断增大。但只要pmax加上待分配结点长度小于等于S，就可以继续进行动态分配。



北京大学信息学院 ©版权所有，转载或翻印必究 Page 67

可利用空间表动态存储区的分配算法

```
void * LinkNode<Elem>::operator new(size_t){
    LinkNode<Elem> * temp;
    if(avail == NULL)      { //可利用空间表为空
          return ::new LinkNode;  //利用系统的new分配空间
          if (pmax + K > S)
                cout<<"没有可分配的空间!";
          else {
                temp = pmax;     pmax = pmax + K;
               return temp;
}}
else{
          temp = avail;      avail = avail->link;
          return temp;
}}
```

北京大学信息学院 ©版权所有，转载或翻印必究 Page 68

算法的缺点

- 这种方法存在着一些严重的问题：每个可利用空间表的结点大小是固定的，后备存储区里的空间一旦被分配就不能再回到后备存储区中
- 如果pmax值已经达到或超过S值而不能再分配空间时，实际上系统中别的可利用空间表中可能还存在大量的空闲结点。

北京大学信息学院 ©版权所有，转载或翻印必究 Page 69

存储的动态分配和回收

- 把各种大小的结点（又称可利用块）组织在一个可利用空间表中；分配时需要按申请的长度在可利用空间表中进行检索，找到其长度大于等于申请长度的结点，从中截取合适的长度
- 回收时也不能简单地把删除的结点放回到可利用空间表中，而必须考虑刚刚被删除的结点空间能否与可利用空间表中的某些结点合并，组成较大的结点，以便能满足后来的较大长度结点的分配请求。

北京大学信息学院 ©版权所有，转载或翻印必究 Page 70

本方法的优缺点

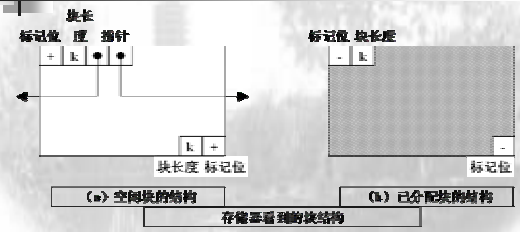
- 这种处理方法的优点是可以解决存储空间的共享问题
- 缺点是分配和回收的算法复杂了，并且在系统长期动态运行的过程中，这种方法有可能使整个空间被分割成许多大小不等的碎块，而某些碎块由于太小而长期得不到使用，这就产生所谓碎片问题。

北京大学信息学院 ©版权所有，转载或翻印必究 Page 71

空闲块的数据结构

- 空闲块的长度是不定的，所以可利用空间表中每个结点都需要记录本结点的长度。
- 一种常见的方法是，对于一个需要m字节空间的请求，存储管理器可能会分配稍多于m字节的空间
- 额外的空间留给存储管理器进行存储管理，例如存放块的标记位、链表指针和块长度。
- 标记位用来区别这个块是空闲块还是已被分配的块。

北京大学信息学院 ©版权所有，转载或翻印必究 Page 72



动态分配

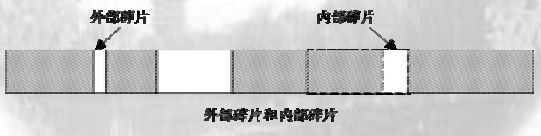
- 动态分配就是要在可利用空间表中找到一个合适的空闲块以满足存储请求
- 即为了分配一个N字节大小的空间，就要利用某种策略从可利用空间表中搜索到一个长度大于等于N字节的空闲块进行分配；如果可利用空间表中不存在这样的结点，就不能进行分配。

分配方法

- 如果能够找到大于N字节的空闲块，一种可能的做法是整个块交给请求者，整个空闲块的首尾标记位都修改为“-”，表示它是已分配块
- 另一种做法是对于一个长度为K（K>N）字节的空闲块，存储管理器把N字节分配给请求者，剩余的K-N字节形成一个新的空闲块；对已分配块和剩余空闲块设置相应的标记、长度和链指针字段。
- 不管采用哪种做法，都会存在所谓碎片(fragmentation)的问题。

碎片类型

- 上述的第一种做法，把多于请求字节数的空间分配给请求者，会浪费空闲块的空间，从而产生“内部碎片”(internal fragmentation)。
- 如果按照前述的第二种做法，把满足请求后的K-N个字节作为新的空闲块，则会产生大量的不能满足一般性请求的小空闲块，称为“外部碎片”(external fragmentation)。



顺序适配(sequential fit)

- 下面先介绍一些可能产生外部碎片的方法，这类方法通称为顺序适配(sequential fit)。
- 常见的顺序适配方法有这几种：
  - 首先适配(first fit)
  - 最佳适配(best fit)
  - 最差适配(worst fit)

首先适配

- 首先适配从可利用空间表的表头开始，顺次在可利用空间表中进行搜索，一旦找到第一个长度大于等于请求块长度的空闲块，就进行分配。
- 一种简单改进是：记住前一次搜索到达的位置，从该位置开始新搜索。当搜索到达链表尾部的时，重新从链表头处开始搜索。
- 首先适配的优点是速度快，缺点是可能把较大块拆分成较小的块，导致后来对大块的申请难以满足。

最佳适配

- 最佳适配在所有长度大于等于请求块长的块中找出最小的一块进行分配。
- 最佳适配的优点：它可以使得无法满足大请求块的可能性降到最低。
- 最佳适配的缺点：使得外部碎片问题变得非常严重，因为那些满足请求后剩下的空闲块非常小，对将来的请求就没有什么用处了。

最佳适配的方案

- 实现最佳适配有两种方案。
  - 其一，检索整个无序的可利用空间表，找到满足分配请求的最小空闲块。
  - 其二，把空闲块按照从小到大顺序排列成优先队列，使得检索时只需要从表头往后查看，直至找到第一块满足分配请求的空闲块；但回收空闲块时，需要插入到优先队列中合适的位置。
- 这两种方案耗时都比较多。最佳适配的结果是空闲块的长度变化很大。

最差适配

- 最差适配采用的是一种与最佳适配完全相反的策略，它在所有长度大于等于请求块长的空闲块中找出最大的一块进行分配。适合于存储分配长度请求比较均匀的情况。
- 实现最差适配有两种方案。
  - 其一，检索整个无序的可利用空间表，找到整个可利用空间的最大空闲块。
  - 其二，把空闲块按照从大到小顺序排列成优先队列，检索时如果表头结点满足分配请求，则把它分配出去，否则存储分配要求无法满足；但回收空闲块时，需要插入到优先队列中合适的位置。

三种方法举例

- 假设可利用空间表中包含三个可利用块，其大小分别是：1200，1000，3000，现有如下一串存储分配的请求：

  600，500，900，2200

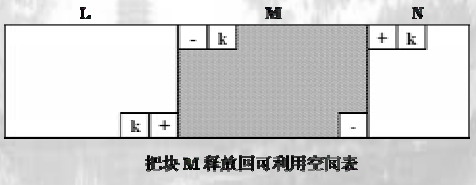
- 采用最佳适配。分配600以后，空闲块为1200，400，3000；分配500以后，空闲块为700，400，3000；分配900以后，空闲块为700，400，2100；再分配2200，发生溢出。

三种方法举例（续）

- 采用首先适配。分配600以后，空闲块为600，1000，3000；分配500以后，空闲块为100，1000，3000；分配900以后，空闲块为，100，100，3000；分配2200以后，空闲块为100，100，800
- 采用最差适配。分配600以后，空闲块为1200，1000，2400；分配500以后，空闲块为1200，1000，1900；分配900以后，空闲块为1200，1000，1000；再分配2200，发生溢出。

回收

- 采用顺序适配方法，当一个块被释放时，需要把它重新链回到可利用空间表中。如果这个块与可利用空间表中其他空闲块物理地址相邻，则需要合并这些相邻块，这样存储管理器才能满足将来尽可能大的存储请求。



把块M释放回可利用空间表

回收过程

- 把块M释放到可利用空间表的过程：
  - 首先检查块M左邻的存储单元L，根据其标记位可以判断块L是不是空闲块。
  - 如果不是，则简单地把M插入可利用空间表。
  - 如果如图中所示，L是空闲块，则把L的长度扩展到包含M。然后，检查M的右邻块N的标记位，如果块N是空闲块，则要把N从可利用空间表中删除，同时扩展M的长度（在本例则是扩展已包含M的L的长度）。
  - 经过对块M的回收操作之后，就形成了一个包含原来的L、M、N三个块长度的大空闲块。

回收的策略

- 很难笼统地讲这哪种适配策略最好。需要考虑以下因素来确定采用哪种顺序适配方案
  - 用户的要求
  - 分配或回收效率对系统的重要性
  - 所分配空间的长度变化范围
  - 分配和回收的频率。
- 在实际应用中，由于首先适配其分配和回收的速度比较快，而且支持比较随机的存储请求，因为应用得更广泛。

伙伴系统

- 伙伴系统的数据块中不再存放任何关于块本身的信息字段。
- 伙伴系统假设存储空间的大小为2^M，M为正整数。当然，这个假设对大多数实际存储器也是成立的。
- 伙伴系统中的每一个空闲块和已分配块的大小都是2^k，k小于等于M。系统为每一种大小的空闲块都单独建立一个列表，系统中保留的列表数目决不会超过M个。

“伙伴”的含义

- 每一个块都有一个对应的“伙伴”块，它们的大小是相同的。
- 对伙伴系统而言，2^k大小的块首地址与它的伙伴的首地址相比，除了第k位之外（最右为第0位），所有的位都相同。
- 例如，长度为$8=2^3$个字节、首地址为0000的块，其伙伴是首地址为1000的块（从右往左数，1000与0000的第3位相同）。

伙伴系统的分配

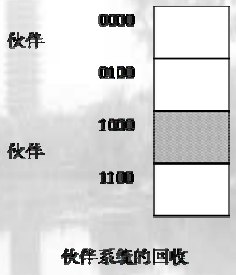
- 对于一个需要N个字节的存储请求，伙伴系统首先确定使得$2^k \geq N$的最小k值。
- 如果在可利用空间表中能找到2^k大小的空闲块，就分配这个空闲块；否则，就找一个更大的空闲块，把它均分成两半；
- 不断重复这个分割的过程，直到生成一个2^k大小的空闲块，并把它分配出去。在分割过程中生成的空闲块都记录到可利用空间表中去。

伙伴系统的特点

- 伙伴系统的一个缺点是它允许内部碎片的存在，因为很难有恰好$2^k=N$的情况发生。但它的外部碎片却非常少。
- 伙伴系统的另一个优点是从可利用空间表中搜索空闲块的速度极快，因为只需要从2^k大小的空闲块中取出第一个就可以了。
- 此外，在回收的时候，合并相邻空闲块也十分简单。

伙伴系统的回收过程

- 长度为$4=2^2$个字节、首地址为1000的块即将被释放回可利用空间中，根据伙伴系统的特点，此块的伙伴是首地址为1100的块。
- 通过搜索对应的块长度表就可以找到伙伴。如果伙伴是空闲块，就把它们进行合并，形成更大的块，按此循环直到不能继续合并。最后再把合并所形成的大空闲块放到可利用空间表中。



伙伴系统的回收

失败处理策略和无用单元回收

- 如果遇到因内存不足而无法满足一个存储请求，存储管理器可以有两种行为：
  - 一是什么都不做，直接返回一个系统错误信息；
  - 二是使用失败处理策略(failure policy)来满足请求。

存储压缩(compact)

- 当使用顺序适配存储分配方法时，可能有足够的空间来满足请求，因为这些方法造成了不少外部碎片，把它们收集起来可能得到连续的大块存储空间能够满足存储请求。这就是存储压缩，通过压缩把内存中的所有碎片集中起来组成一个大的可利用块。
- 只有内存碎片很多却分配不出有用空间而即将产生溢出时，不得已才使用

句柄(handle)

- 存储压缩之后，应用程序的数据地址将发生变化。如果应用程序以某种方式依赖于数据的绝对位置，就会引起严重的后果。
- 句柄是对存储位置的二级间接指引，可以使得存储地址相对化。存储管理器在进行内存分配的时候不是返回一个指向空闲块的指针，而是返回一个指向变量的指针，这个变量才指向存储位置，这个变量就是句柄。
- 可以移动存储块的位置，而只需要修改句柄的值，不需要修改应用程序。

无用单元收集(garbage collection)

- 最彻底的失败处理策略是无用单元收集。所谓无用单元是指那些可以回收而没有回收的结点空间。
- 高级程序设计语言中，程序员常犯的一种错误就是动态生成变量或对象并使用它们之后，忘记了释放其内存空间，以后却又不再使用它们。也可能是由于程序要做到及时释放无用单元有困难。
- 这种丢失的存储空间称为无用单元(garbage)，也称为内存泄漏(memory leak)

回收无用结点

- 这些无用结点的存在会影响空间使用的效率。因此存储管理系统要有能力把这些无用结点找出来，并送回到可利用空间表中去。
- 通常的作法是：首先普查一次内存，把那些已经不属于任何链上的结点打上标志，然后将它们收集到可利用空间表中，回收过程通常还可与存储压缩一起进行。

无用单元收集的算法

- 首先介绍引用计数(reference count)算法。
- 系统为每一个动态分配的存储块加入一个计数字段。当一个新指针指向该存储块时，存储块的引用计数就会加1；当某指针不再指向这个块时，其引用计数就会减1。
- 一旦存储块的引用计数变为0，这个存储块成为无用单元，立即放回到可利用空间表中。

引用计数算法的优缺点

- 优点：不需要一个明显的无用单元收集阶段，每当存储单元成为无用单元就立即把它放到可利用空间表中。
- 缺点：首先，必须为每一个存储对象维护一个引用计数。如果有很多较小的对象，系统的额外负担太多。当存在无用单元循环引用时，就会出现另一个严重问题：每个存储对象都被指向一次，但是对象集合仍然是无用单元，因为没有指针指向对象集合。

标记/清除(mark/sweep)方法

标记/清除算法的实质就是周游广义表。

- 在这种方法中，每一个存储对象只需要一个简单的标记位。一旦可利用空间表用完，存储管理器就会进入一个独立的无用单元收集阶段。
- 首先清除所有的标记位，然后从变量表中的每一个变量开始，沿着指针进行深度优先搜索。每遇到一个存储单元，就设置其标记位为“已访问”。
- 最后访问所有存储单元，对存储区域进行清理——所有未标记的单元就认为是无用单元，可以释放到可利用空间表中。

标记/清除方法的优缺点

- 标记/清除方法的优点是它比引用计数方法需要的空间少，而且对于循环情况也能工作。
- 但它隐藏了一个很大的缺陷，这就是进行处理所需要的空间需求。周游广义表的算法实质上是一个递归算法，在这种情况下编译器的运行系统维护一个栈；要么存储管理器维护它自己的栈。而无用单元收集算法往往是在内存空间很紧张时进行的，必须尽可能地采用最少的空间代价。

Deutsch-Schorr-Waite无用单元收集算法

- Deutsch-Schorr-Waite无用单元收集算法采用逆转链的广义表周游算法，而不需要额外的用于栈的空间。
- 在周游中每深入一步，不需要向栈中存储一个指针，而是借用即将深入的那个指针，把它逆转设置为指向前一步刚经过的结点

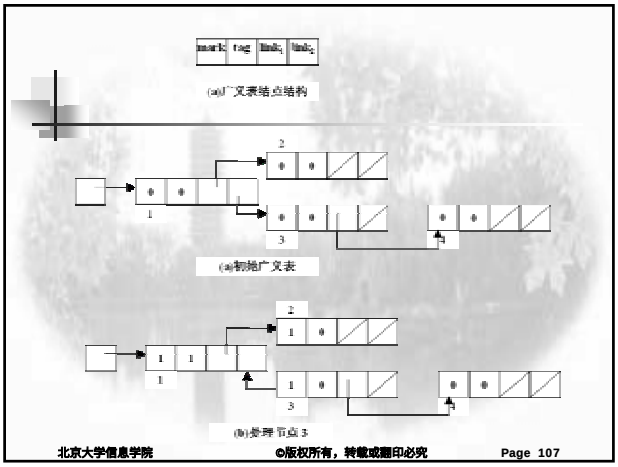
暂时借用外存空间

- 可以将正在使用的单元复制到临时外存数据块中
  - 需要注意更新相应的指针和引用信息
- 新腾出的内存可以被操作系统用于标记/清除算法需要的临时空间

适合于虚拟存储计算机系统

Deutsch-Schorr-Waite无用单元收集算法

- 算法的关键是设置两个标志位mark和tag。
- mark初值为“UNVISITED”(即“0”)，执行标志算法后，将全部有用结点的mark字段置为“VISITED”（已访问）。tag初值也为0，在标志算法执行过程中，周游到本结点所对应的子表时，被置成“1”，返回以后再恢复成“0”。
- 前进时不仅指针的值要求非空，同时还要求指针所指的结点mark字段为“UNVISITED”（未访问），后退时则只要求指针为空或者指针所指的结点mark字段为“VISITED”



Deutsch-Schorr-Waite算法的主要缺点

- Deutsch-Schorr-Waite算法的主要缺点是在执行过程中要修改表中指针的值，所以不能再入运行。在执行完无用单元的收集算法以后，只要扫描一遍内存，将所有mark字段为“UNVISITED”（未访问）的结点送入可利用空间表中。