第6章 广义表

- 6.1 广义表的基本概念
- 6.2 广义表的存储结构
- 6.3 广义表的操作算法

6.1 广义表的基本概念

- 广义表（列表）的概念
  - $n(\geq 0)$个表元素组成的有 限序列，记作

    $LS=(a_1, a_2, a_3, \ldots, a_n)$

    *LS*是表名，a_i是表元素，它可以是单个元素(称为原子)，可以是广义表(称为子表)。

- n为表的长度。$n=0$的广义表为空表。

- $n>0$时，表的第一个表元素称为广义表的表头(*head*)，除此之外，其它表元素组成的表称为广义表的表尾(*tail*)。

广义表举例：

（1）A=（）

（2）B=（e）

（3）C=（a, (b, c, d))

（4）D=（A，B，C）

（5）E= (a , E）

- ✓ 任意一个非空广义表，均可分解为表头和表尾。
- ✓ 对于一个非空广义表，其表头可能是原子，也可能是子表；而表尾一定是子表。

广义表的特性

$A = ()$

$B = (6, 2)$

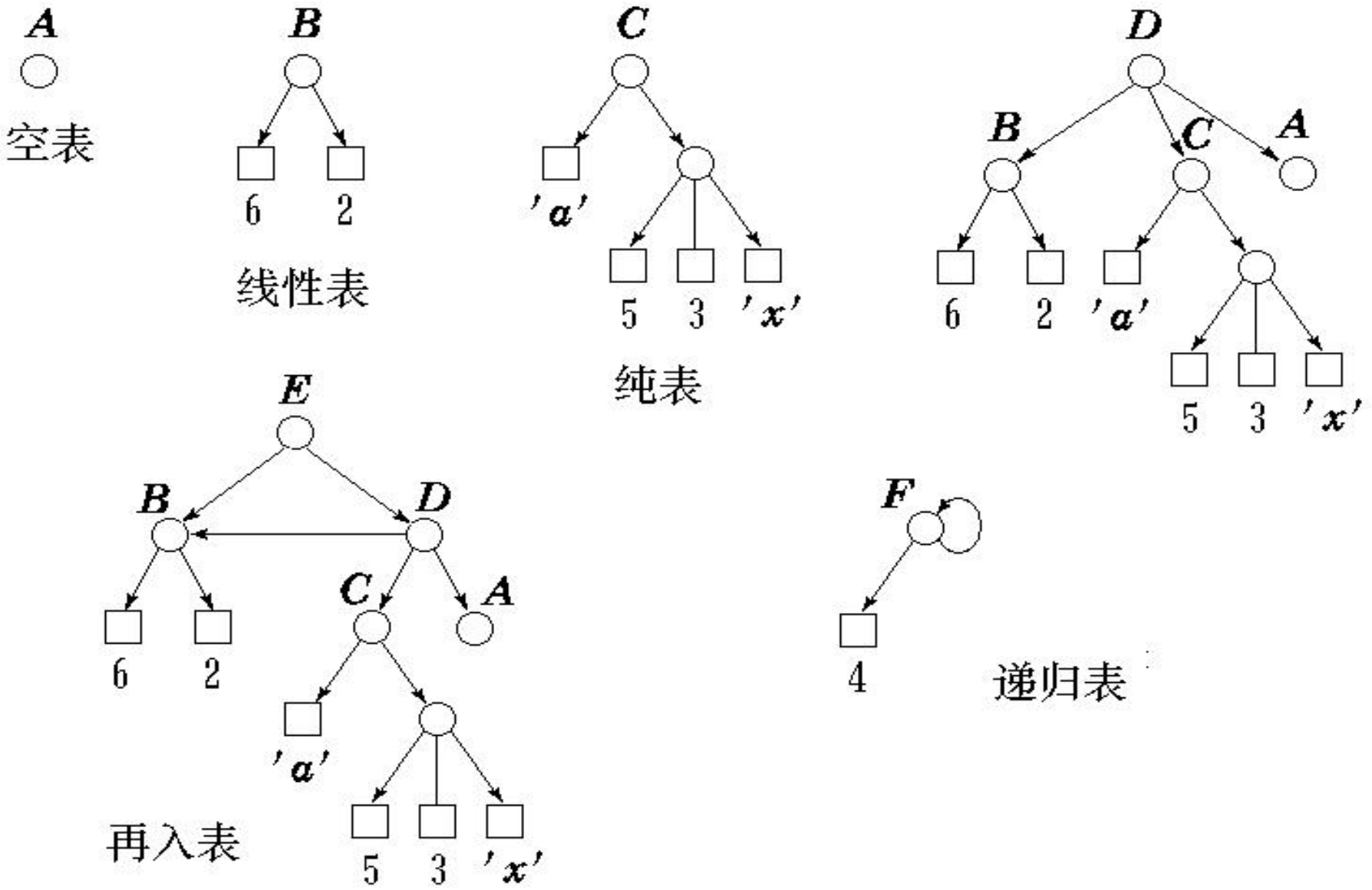
$C = ('a', (5, 3, 'x'))$

$D = (B, C, A)$

$E = (B, D)$

$F = (4, F)$

- 广义表是一个多层次结构；
- 广义表的深度定义为所含括弧的重数；
  注意：“原子”的深度为0；
  “空表”的深度为1
- 广义表可以共享；
- 广义表可以是一个递归的表；



各种广义表的示意图

广义表的基本操作：

- 结构的创建和销毁

  InitGList(&L);　　DestroyGList(&L);

  CreateGList(&L, S);　　CopyGList(&T, L);

- 状态函数

  GListLength(L);　　GListDepth(L);

  GListEmpty(L);　　GetHead(L);　　GetTail(L);

- 插入和删除操作

  InsertFirst_GL(&L, e);

  DeleteFirst_GL(&L, &e);

- 遍历

  Traverse_GL(L, Visit());

6.2 广义表的存储结构

- 能否采用顺序存储结构？
  - 由于广义表中的元素不是同一类型，因此难以用顺序结构表示，通常采用链接存储方法存储广义表，并称之为广义链表。
- 由于广义表中有两种数据元素，原子或子表，因此，需要两种结构的结点：
  - 原子结点
  - 子表结点
- 下面介绍一种广义表的链式存储结构。

扩展的线性链表表示法：

- 子表结点由三个域组成：
  - 标志域（type）
  - 子表头指针域（sublist）
  - 指向下一个元素的指针域（next）

子表结点

- 原子结点的三个域为
  - 标志域（type）
  - 值域（data）
  - 指向下一个元素的指针域（next）

原子结点

其类型定义如下：

```
Enum GListNodeType{ ATOM, LIST };

template <class T>
struct GListNode{
      GListNodeType  type;
      union{
          T data;
           GListNode<T> *sublist;    //表结点的表头指针
        };
       GListNode<T>  *next;   //指向下一个元素结点
};
```

广义表的存储结构示例：

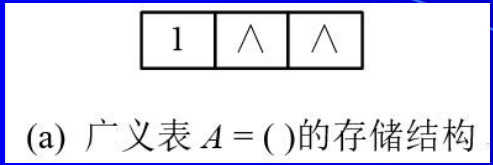
(a) $A = ( )$

(b) $B = (a, b, c)$

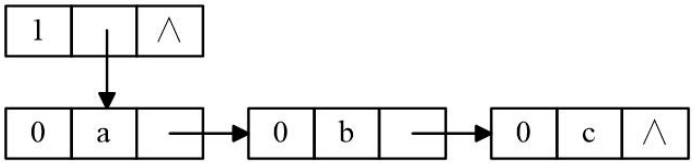
(c) $C = (a, (b, c, d), e)$

(d) $D = ((a, b), c, (d, (e, f), g))$

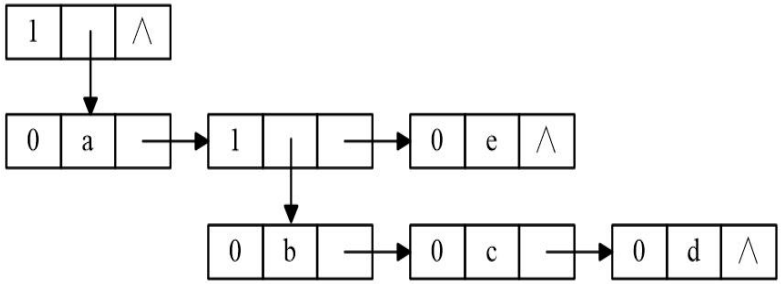
(e) $E = (a, ( ), (( ), ( )), b)$



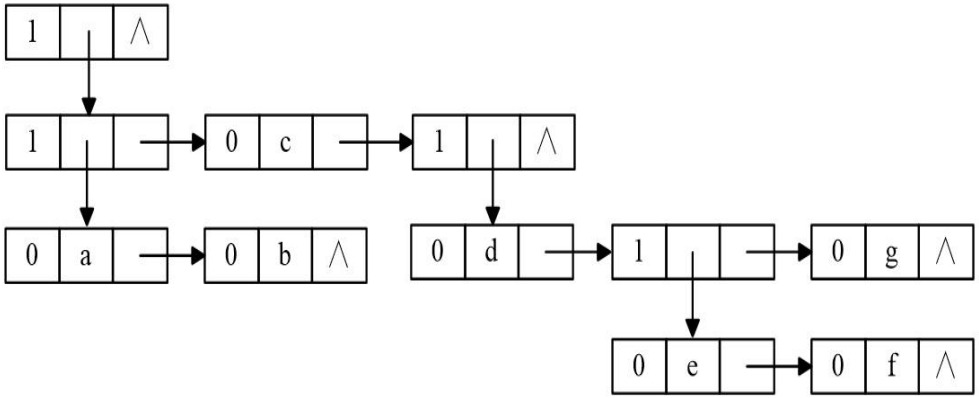
(a) 广义表 $A=(\ )$的存储结构



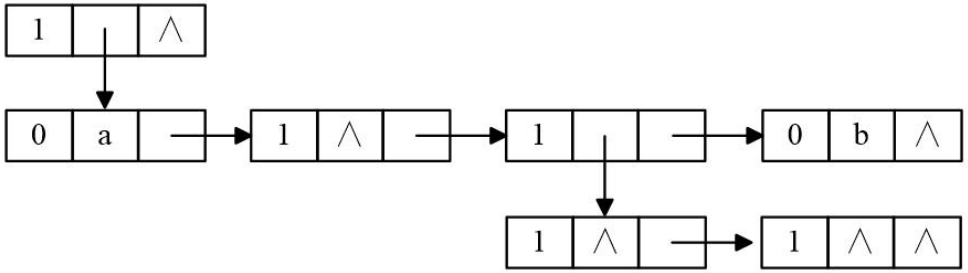
(b) 广义表 B = (a, b, c)的存储结构



(c) 广义表 C = $(a, (b, c, d), e)$的存储结构



(d) 广义表 D = $((a, b), c, (d, (e, f), g))$的存储结构



(e) 广义表 E = $(a, (\ ), ((\ ), (\ )), b)$的存储结构

广义表类定义：

```
template <class T>
class GList{
   GListNode<T> *head;
public:
   GList();
   GList(GList<T> head, GList<T> tail);
   GList(const GList<T> &gl);    // 拷贝构造函数
    ~GList();
    void Traverse(); // 遍历算法
    int Length();   // 计算表的长度
    int Depth();    // 计算表的深度
    ……
};
```

6.3 广义表的操作算法

- 递归函数的概念

  一个含直接或间接调用本函数语句的函数被称之为递归函数，

- 递归函数必须满足以下两个条件

  1) 在每一次调用自己时，必须是(在某种意义上)更接近于解;

  2) 必须有一个终止处理或计算的准则。

- 递归设计的两个基本要素
  - 归纳项
  - 终结条件

递归设计的常见策略

- 直接递归法
  - 按问题或结构的定义直接给出递归的归纳项
  - 特别适用于结构上可以分解的结构，
    如:广义表、二叉树、树等
- 分治法（ **Divide and Conquer** ）
- 减治法（ **Decrease and Conquer** ）
- 变治法（ **Transform and Conquer** ）

分治法 (Divide and Conquer)

- 对于一个输入规模为n的问题：
  - 用某种方法把输入分割成k(1<k≤n)个子集，从而产生k个子问题；
  - 分别求解这k个问题，得出k个问题的子解；
  - 再用某种方法把它们组合成原来问题的解。
  - 若子问题还相当大，则可以反复使用分治法，直至最后所分得的子问题足够小，以至可以直接求解为止。

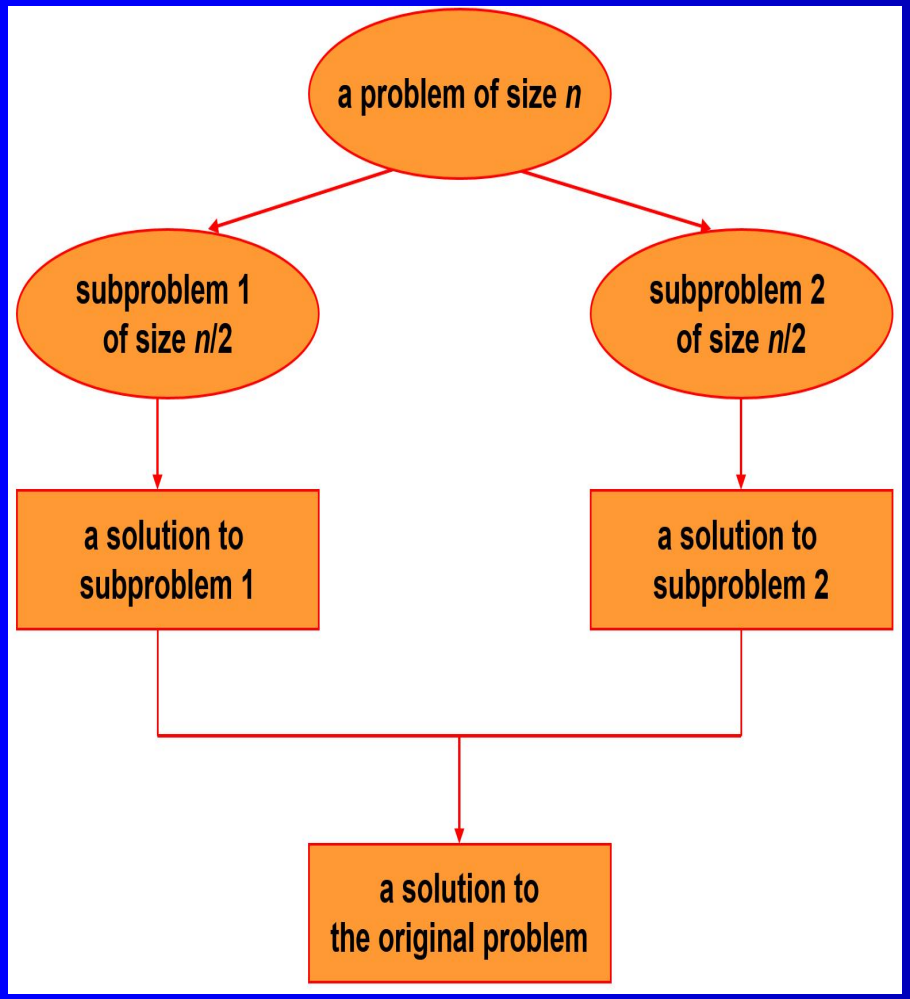
孙子曰：凡治众如治寡，分数是也

减治法 (Decrease and Conquer)

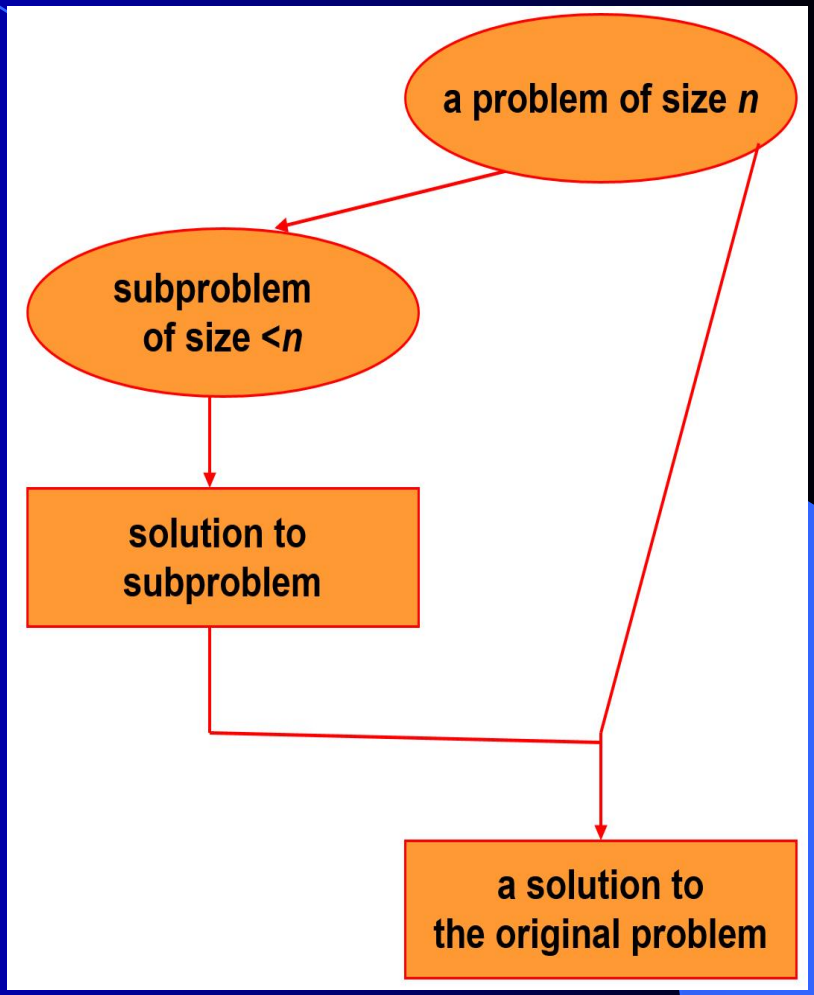
- 萨特斯为了告诉他的士兵坚忍和智慧比蛮力更重要的道理，把两匹马带到他们面前，然后让两个人拔光马尾上的毛。一个人是魁梧的大力士，他直接抓住马尾拔了又拔，但一点效果都没有；另一个人是一个精明的、长相狡猾的小裁缝，他微笑着，每次拔掉一根毛，很快就把马尾巴拔的光秃秃的。

——E. Cobham Brewer

分治法


a problem of size n
subproblem 1 of size n/2
subproblem 2 of size n/2
a solution to subproblem 1
a solution to subproblem 2
a solution to the original problem

减治法


a problem of size n
subproblem of size <n
solution to subproblem
a solution to the original problem

几种不同递归方法的差异

- 计算 a^n

- Brute Force: $a^n = a*a*a*a*\ldots\ *a$
- Divide and conquer: $a^n = a^{n/2} * a^{n/2}$
- Decrease by one: $a^n = a^{n-1} * a$
- Decrease by constant factor $a^n = (a^{n/2})^2$