

## 数据结构与算法精讲<sup>v1.2</sup>

太原联航精英训练营

1

1

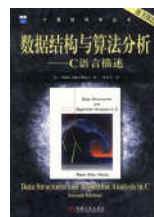
## 第一章 绪论

2

2

## 参考书目

《数据结构（C语言版）》 严蔚敏	清华大学出版社
《数据结构与算法分析：C语言描述》 Mark Allen Weiss	机械工业出版社
《算法设计与分析》（第二版） 王晓东	清华大学出版社
《算法导论》（第三版） Thomas H.Cormen	机械工业出版社
《算法设计与分析基础》（第3版） Anany Levitin	电子工业出版社
《算法》（第4版） Robert Sedgewick	人民邮电出版社



## 绪论

目前，计算机已深入到社会生活的各个领域，其应用已不再仅仅局限于**科学计算**，而更多的是用于**控制**，**管理**及**数据处理**等**非数值计算**领域。计算机是一门研究用计算机进行信息表示和处理的科学。这里面涉及到两个问题：**信息的表示**，**信息的处理**。

信息的表示和组织又直接关系到处理信息的程序的效率。随着应用问题的不断复杂，导致信息量剧增与信息范围的拓宽，使许多系统程序和应用程序的规模很大，结构又相当复杂。因此，必须分析待**处理问题中的对象的特征及各对象之间存在的关系**，这就是数据结构这门课所要研究的问题。

## 计算机求解问题的一般步骤

编写解决实际问题的程序的一般过程：

- 如何用数据形式描述问题?—即由问题抽象出一个适当的数学模型;
- 问题所涉及的数据量大小及数据之间的关系;
- 如何在计算机中存储数据及体现数据之间的关系?
- 处理问题时需要对数据作何种运算?
- 所编写的程序的性能是否良好?

上面所列举的问题基本上由这门课程来回答。

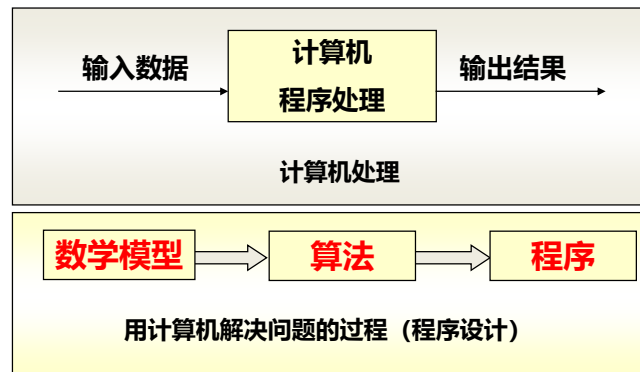


## 为何要学？学什么？学习目标？

- 数据结构      在计算机相关专业课程体系中，一直处于**核心**位置  
是计算机科学的重要组成部分  
是设计与实现高效算法的基石
- 讲授范围      各类数据结构设计和实现的基本原理与方法  
算法设计和分析的主要技巧与工具
- 学习数据结构就是要  
学会高效地利用计算机，有效地存储、组织、传递和转换数据  
掌握各类数据结构功能、表示、实现和基本操作接口  
理解各类基本算法与不同数据结构之间的内在联系  
了解各类数据结构适用的应用环境  
灵活地选用各类基本算法及对应的数据结构，解决实际问题



## 数据结构的定义



**程序设计:** 为计算机处理问题编制一组指令集

**算法:** 处理问题的策略

**数据结构:** 数据的组织与操作

7

7

## 数据结构的定义

- Niklaus Wirth (尼古拉斯·沃斯) :
  - Algorithm + Data Structures = Programs

Algorithm ['ælgərɪðəm]



- Data Structures =
  - Data Set + Relations + Operations

数据集 + 关系 + 操作

8

8

## 数据结构的基本概念

**数据(Data)**：是客观事物的符号表示。在计算机科学中指的是所有能输入到计算机中并被计算机程序处理的符号的总称。

**数据元素(Data Element)**：是数据的基本单位，在程序中通常作为一个整体来进行考虑和处理。

一个数据元素可由若干个**数据项(Data Item)**组成。数据项是数据的不可分割的最小单位。数据项是对客观事物某一方面特性的数据描述。

**数据对象(Data Object)**：是性质相同的数据元素的**集合**，是数据的一个子集。如字符集合 $C=\{'A','B','C',...\}$ 。

## 数值计算问题举例

数值计算的程序设计**问题**

例：物体从100米高的塔顶落到地面的时间——二次方程

**建模：**

涉及对象：高度 $h$ ，时间 $t$ ，重力加速度 $g$  ( $g=9.8$ )

对象之间的关系： $h = \frac{1}{2}gt^2$

设计求解问题的方法： $t = \text{sqrt}(2h/g)$

**编程：**

```
int main ( )
{
    float t, h, g;
    g = 9.8;
    scanf ("%f", &h);
    t = sqrt(2 * h / g);
    printf ("The falling time is %f\n", t);

    return 0;
}
```

## 非数值计算的程序设计问题

### 例1：求一组整数(假设5个)中的最大值

#### 建模：

涉及对象：5个整数

对象之间的关系：大小关系

#### 设计求解问题的方法：

基本操作是“比较两个数的大小”

首先将第一个数记为当前最大值，然后依次比较其余n-1个整数，如果该某个整数大于当前最大值，就更新当前最大值。

#### 编程：

11



11

## 非数值计算的程序设计问题

### 例：求一组整数(5个)中的最大值

```
int main ( )
{
    int d[5], i, max;

    for( i=0; i<5; i++)
    {
        scanf ("%d", &d[i]);
    }
    for( i=1; i<5; i++)
    {
        if ( max < d[i])
        {
            max = d[i];
        }
        printf ("The max number is %f\n", max);
    }

    return 0;
}
```

12



12

## 数据结构研究的主要内容

### ■ 概括地说：

- 数据结构是一门研究“非数值计算的程序设计问题中计算机操作对象以及它们之间的**关系**和**操作**”的学科。

### ■ 具体地说：

- 数据结构主要研究数据之间有哪些结构关系，如何**表示**，如何**存储**，如何**处理**。

## 数据结构例子

数据之间结构关系：是具体关系的抽象。

### 例1：

学生间学号顺序关系是一种**线性结构**关系

**线性结构关系**是对学生间学号顺序关系的一种抽象表示

学号	姓名	出生日期	入学日期	班级	专业
cs001	张扬	02121990	01092009	cs20091	计算机
cs002	李明	07111990	01092009	cs20091	计算机
cs003	杨华	01051990	01092009	cs20091	计算机
cs004	贾茹	15041990	01092009	cs20091	计算机

## 数据结构例子

### ■ 例1：定义学生数据结构中的运算：

- 查询学生信息
- 插入学生信息
- 修改学生信息
- 删除学生信息

学号	姓名	出生日期	入学日期	班级	专业
cs001	张扬	02121990	01092009	cs20091	计算机
cs002	李明	07111990	01092009	cs20091	计算机
cs003	杨华	01051990	01092009	cs20091	计算机
cs004	贾茹	15041990	01092009	cs20091	计算机

15

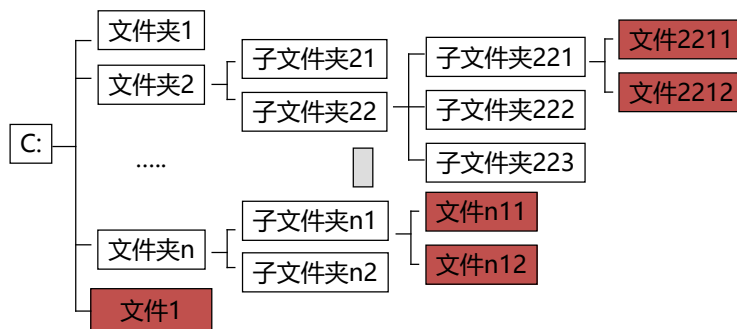
15

## 数据结构例子

### ■ 例2：计算机文件系统

文件夹或文件间的关系是一种树型结构关系

树型结构关系是对文件系统关系的一种抽象表示



16

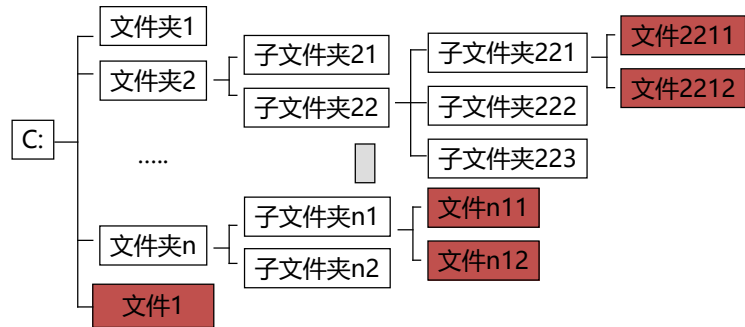
16



## 数据结构例子

### ■ 例2：定义计算机文件系统中的操作

- 查找、新建、删除文件夹
- 查找、新建、删除文件



17

17

## 数据结构例子

### ■ 例3：地铁线路图



18

18

## 数据结构例子

- 例3：地铁线路规划
- 定义地铁图中的操作：
  - 查询地铁站
  - 规划乘车路线
  - 增加地铁站
  - 关闭地铁站...

可见，不同的“关系”构成不同的“数据结构”。

这些关系称为数据的逻辑结构。

数据结构是相互之间存在着某种逻辑关系的数据元素集合  
及定义在该集合上的操作集合

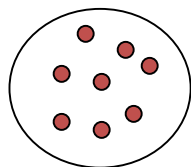
19

19

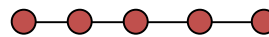
## 数据的逻辑结构

- 数据的逻辑结构  
—— 描述数据间的逻辑关系

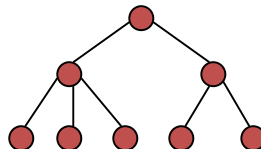
- 数据的逻辑结构可归结为以下四类:



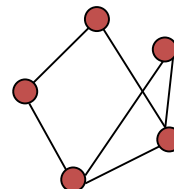
a. 集合关系



b. 线性关系



c. 树型关系



d. 图型关系

20

20

## 数据的逻辑结构

**数据结构(Data Structure)**: 是指相互之间具有(存在)一定联系(关系)的数据元素的集合。**元素之间的相互联系(关系)称为逻辑结构。**

数据元素之间的关系可以是元素之间代表某种含义的自然关系,也可以是为处理问题方便而人为定义的关系,这种自然或人为定义的“关系”称为数据元素之间的逻辑关系,相应的结构称为逻辑结构。

数据元素之间的逻辑结构有四种基本类型,如上图所示。

- ① **集合**: 结构中的数据元素除了“同属于一个集合”外,没有其它关系。
- ② **线性结构**: 结构中的数据元素之间存在一对一的关系。
- ③ **树型结构**: 结构中的数据元素之间存在一对多的关系。
- ④ **图状结构或网状结构**: 结构中的数据元素之间存在多对多的关系。

## 数据的存储结构(物理结构)

### ■ 数据的存储结构

—— 逻辑结构在存储器中的映象

### ■ 存储结构包含两个方面:

1. “数据元素”的映象
2. “关系”的映象

## 数据的存储结构－关系的映象

- 即如何表示两个元素之间的关系  $\langle x, y \rangle$

方法一：顺序映象（顺序存储结构）

用数据元素在存储器中的**相对位置**来表示数据元素之间的逻辑关系

方法二：链式映象（链式存储结构）

以**附加信息(指针)**表示数据关系

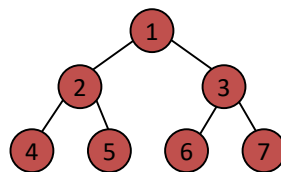
需要用一个和  $x$  在一起的**附加信息**指示  $y$  的存储位置

23

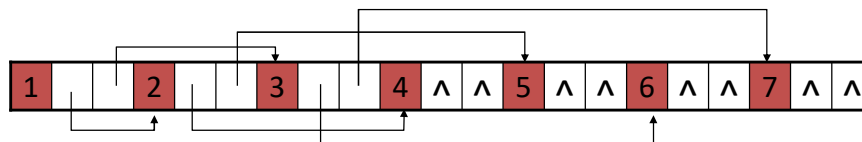
23

## 数据的存储结构－关系的映象

- 树型结构的链式存储结构



树型结构



24

24

## 存储结构

### 1. 顺序存储

用数据元素在存储器中的相对位置来表示数据元素之间的逻辑结构(关系)。

### 2. 链式存储

在每一个数据元素中增加一个存放另一个元素地址的指针(pointer)，用该指针来表示数据元素之间的逻辑结构(关系)。

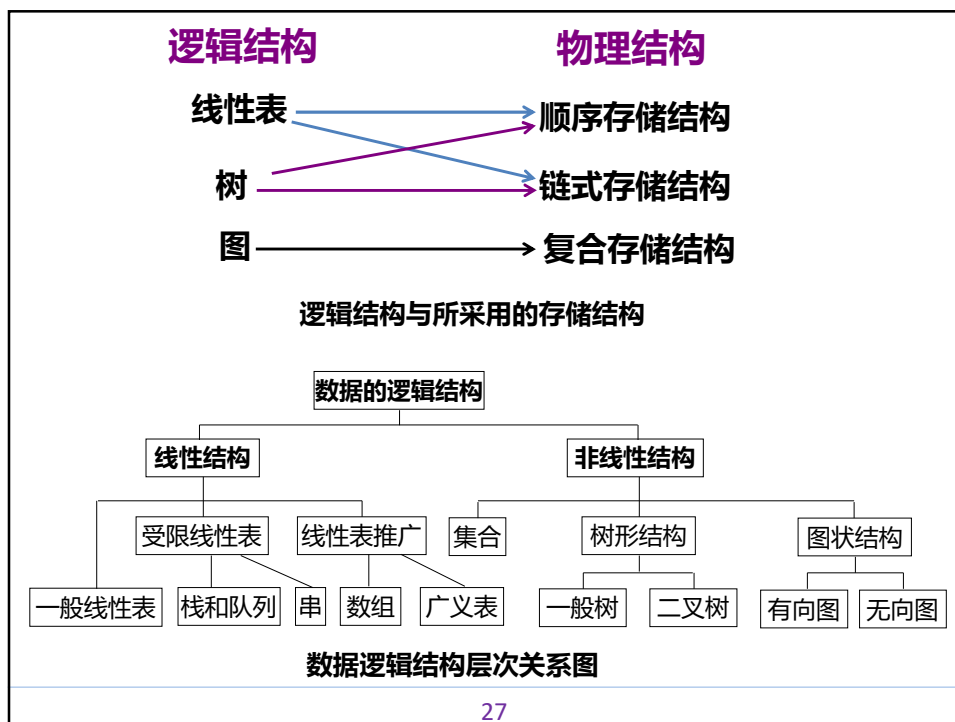
在C语言中，用**一维数组**表示顺序存储结构；用**结构体类型**表示链式存储结构。

## 逻辑结构和物理结构的关系

- 数据的逻辑结构和物理结构是密不可分的两个方面，一个算法的**设计**取决于所选定的**逻辑结构**，而算法的**实现**依赖于所采用的**存储结构**。

### ■ 数据结构的三个组成部分：

1. **逻辑结构**：数据元素之间逻辑关系的描述
2. **存储结构**：数据元素在计算机中的存储及其逻辑关系的表现称为数据的存储结构或物理结构。
3. **数据操作**：对数据要进行的运算。




27

## 数据结构的运算

数据结构的主要运算包括：

- (1) **建立(Create)**一个数据结构；
- (2) **消除(Destroy)**一个数据结构；
- (3) 从一个数据结构中**删除>Delete)**一个数据元素；
- (4) 把一个数据元素**插入**(Insert)到一个数据结构中；
- (5) 对一个数据结构进行**访问**(Access)；
- (6) 对一个数据结构(中的数据元素)进行**修改**(Modify)；
- (7) 对一个数据结构进行**排序**(Sort)；
- (8) 对一个数据结构进行**查找**(Search)。

28

 **联航精英训练营**

28

## 数据类型

**数据类型(Data Type)**: 指的是一个值的集合和定义在该值集上的一组操作的总称。

数据类型是和数据结构密切相关的一个概念。在C语言中数据类型有：**基本类型和构造类型**。

数据结构不同于数据类型，也不同于数据对象，它不仅要描述数据类型的数据对象，而且要描述数据对象各元素之间的相互关系。

## 抽象数据类型

**抽象数据类型**(Abstract Data Type, 简称ADT): 是指一个数学模型以及定义在该模型上的一组操作。

ADT的定义仅是一组逻辑特性描述，与其在计算机内的表示和实现无关。因此，不论ADT的内部结构如何变化，只要其数学特性不变，都不影响其外部使用。

ADT的形式化定义是三元组： $ADT=(D, S, P)$

其中：D是**数据对象**，S是D上的**关系集**，P是对D的**基本操作集**。

## 抽象数据类型

ADT的一般定义形式是：

```
ADT <抽象数据类型名>{  
    数据对象： <数据对象的定义>  
    数据关系： <数据关系的定义>  
    基本操作： <基本操作的定义>  
} ADT <抽象数据类型名>
```

- 其中数据对象和数据关系的定义用伪码描述。
- 基本操作的定义是：
  - <基本操作名>(<参数表>)
  - 初始条件： <初始条件描述>
  - 操作结果： <操作结果描述>

**初始条件：**描述操作执行之前数据结构和参数应满足的条件; 若不满足，则操作失败，返回相应的出错信息。

**操作结果：**描述操作正常完成之后，数据结构的变化状况和应返回的结果。

## ADT 的特征

### 特征一：数据抽象

用ADT描述程序处理的实体时，强调的是**其本质的特征**、**其所能完成的功能**以及它和**外部用户的接口**（即外界使用它的方法）。

### 特征二：数据封装

将实体的外部特性和其内部实现**细节分离**，并且对外部用户**隐藏其内部实现细节**。



## ADT 的实现

- ADT需要通过**固有数据类型**(高级编程语言中已实现的数据类型)来实现
- 例如对上述复数ADT的实现:

```
//存储结构的定义
typedef struct
{
    float realpart;
    float imagpart;
}complex;
```

```
//基本操作的函数原型说明
// 构造复数 Z,其实部和虚部分别被赋以参数realval 和 imagval 的值
void Assign( complex &Z, float realval, float imagval );
// 返回复数 Z 的实部值
float GetReal( complex Z );
// 返回复数 Z 的虚部值
float Getimag( complex Z );
// 以 sum 返回两个复数 z1, z2 的和
void add( complex z1, complex z2, complex &sum );
```

33

33

## ADT 的实现 (续)

```
// -----基本操作的实现
// 以 sum 返回两个复数 z1, z2 的和
void add( complex z1, complex z2, complex &sum )
{
    sum.realpart = z1.realpart + z2.realpart;
    sum.imagpart = z1.imagpart + z2.imagpart;
}
```

34

34

## 抽象数据类型与算法

### 抽象数据类型带给算法设计的好处有：

- (1) 算法顶层设计与底层实现分离；
- (2) 算法设计与数据结构设计隔开，允许数据结构自由选择；
- (3) 数据模型和该模型上的运算统一在ADT中，便于空间和时间耗费的折衷；
- (4) 用抽象数据类型表述的算法具有很好的可维护性；
- (5) 算法自然呈现模块化；
- (6) 为自顶向下逐步求精和模块化提供有效途径和工具；
- (7) 算法结构清晰，层次分明，便于算法正确性的证明和复杂性的分析。

## 嵌入式系统软件中数据结构的特点

1. 数据规模小
2. 采用简单的数据结构
3. 采用RAM资源占用比较少的算法
4. 采用程序代码简单的算法

专业铸就品质 梦想成就未来

The Specialty Casts Quality, the Dream Gains Future.

## 第二章 算法的基本概念和算法的衡量

## 算法的特征

**算法(Algorithm)**: 是对特定问题求解方法(步骤)的一种描述,是指令的有限序列,其中每一条指令表示一个或多个操作。

### 算法具有以下五个特性

- ① **有穷性**: 一个算法必须总是在执行有穷步之后结束,且每一步都在有穷时间内完成。
- ② **确定性**: 算法中每一条指令必须有确切的含义。不存在二义性。且算法只有一个入口和一个出口。
- ③ **可行性**: 一个算法是能行的。即算法描述的操作都可以通过已经实现的基本运算执行有限次来
- ④ **输入**: 一个算法有零个或多个输入,这些输入取自于某个特定的对象集合。
- ⑤ **输出**: 一个算法有一个或多个输出,这些输出是同输入有着某些特定关系的量。

一个算法可以用多种方法描述,主要有: **使用自然语言描述; 使用形式语言描述; 使用计算机程序设计语言描述。**

**算法和程序是两个不同的概念。**一个计算机程序是对一个算法使用某种程序设计语言的具体实现。算法必须可终止意味着不是所有的计算机程序都是算法。

39



39

## 算法的评价

评价一个好的算法有以下几个标准

- 1. **正确性(Correctness)**: 算法应满足具体问题的需求。
- 2. **可读性(Readability)**: 算法应容易供人阅读和交流。可读性好的算法有助于对算法的理解和修改。
- 3. **健壮性(Robustness)**: 算法应具有容错处理。当输入非法或错误数据时,算法应能适当地作出反应或进行处理,而不会产生莫名其妙的输出结果。
- 4. **通用性(Generality)**: 算法应具有一般性,即算法的处理结果对于一般的数据集合都成立。
- 5. **效率与存储量需求**: 效率指的是算法执行的时间;存储量需求指算法执行过程中所需要的最大存储空间。一般地,这两者与问题的规模有关。(时间复杂度与空间复杂度)

40



40

## 算法的“正确性”

- 对算法是否“**正确**”的理解可以有以下四个层次：
  - a. 程序中不含语法错误；
  - b. 程序对于几组输入数据能够得出满足要求的结果；
  - c. 程序对于精心选择的、典型、苛刻且带有刁难性的几组输入数据能够得出满足要求的结果；**
  - d. 程序对于一切合法的输入数据都能得出满足要求的结果；
- 通常以**第 c 层**意义的正确性作为衡量一个算法是否合格的标准。

## 算法效率的度量

算法执行时间需通过依据该算法编制的程序在计算机上运行所消耗的时间来度量。其方法通常有两种：

**事后统计：**计算机内部进行执行时间和实际占用空间的统计。

问题：必须先运行依据算法编制的程序；依赖软硬件环境，容易掩盖算法本身的优劣；没有实际价值。

**事前分析：**求出该算法的一个时间界限函数。

算法分析感兴趣的不是具体的资源占用量，而是与具体的平台无关、具体的输入实例无关，且随输入规模增长的值是可预测的。

与此相关的因素有：

- 依据算法选用何种策略；
- 问题的规模；
- 程序设计语言；
- 编译程序所产生的机器代码的质量；
- 机器执行指令的速度；

撇开软硬件等有关因素，可以认为一个特定算法“**运行工作量**”的大小，只依赖于问题的规模（通常用 $n$ 表示），或者说，它是**问题规模的函数**。

## 渐进分析：大O记号

算法中**基本操作重复执行的次数**是问题规模 $n$ 的某个函数，其时间量度记作 $T(n)=O(f(n))$ ，称作算法的**渐近时间复杂度**(Asymptotic Time complexity)，简称**时间复杂度**。

渐进分析：在问题规模足够大后，计算成本如何增长？

当 $n \gg 2$ 后，对于规模为 $n$ 输入，算法

需执行的基本操作次数： $T(n) = ?$

需占用的存储单元数： $S(n) = ?$

一般地，常用**最深层循环内**的语句中的原操作的**执行频度**(重复执行的次数)来表示。

### 大O记号 (big-O notation) (渐近上界O)

“O”表示法的一般提法是：当且仅当存在正整数 $c$ 和 $n_0$ ，使得 $T(n) \leq cf(n)$ 对于所有的 $n \geq n_0$ 成立，则称该算法的时间增长率在 $O(f(n))$ 中，记为 $T(n) = O(f(n))$ 。

与 $T(n)$ 相比， $f(n)$ 更为简洁，但依然反映前者的增长趋势

常数可忽略： $O(f(n)) = O(c \cdot f(n))$

低次项可忽略： $O(na + nb) = O(na)$ ,  $a > b > 0$

43



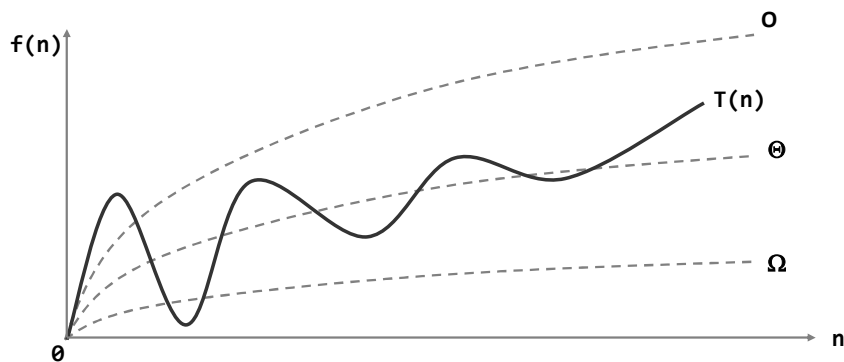
联航精英训练营

43

## 渐进分析：其它记号

- **渐近下界 $\Omega$** ： $T(n) = \Omega(f(n))$ ：  
 $\exists c > 0, n \gg 2$ 后， $T(n) > c \cdot f(n)$

- **同阶 $\Theta$** ： $T(n) = \Theta(f(n))$ ：  
 $\exists c_1 > c_2 > 0, n \gg 2$ 后， $c_1 \cdot f(n) > T(n) > c_2 \cdot f(n)$



44



联航精英训练营

44

## 大O记号常见的表示时间复杂度的阶

- 1) 常量阶:  $O(1)$
- 2) 对数阶:  $O(\log n)$
- 3) 线性阶:  $O(n)$
- 4) 线性对数阶:  $O(n \log n)$
- 5) 平方阶:  $O(n^2)$
- 6) 立方阶:  $O(n^3)$
- 7) 指数阶:  $O(2^n)$
- 8) 阶乘阶:  $O(n!)$
- 9)  $n^n$

以下六种计算算法时间的多项式是最常用的。

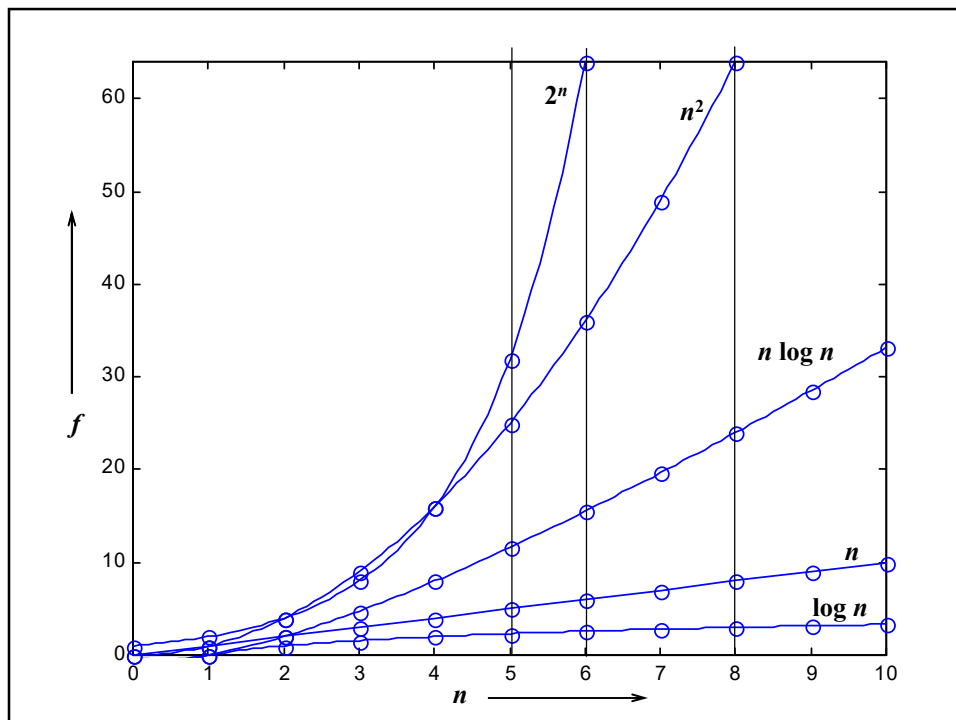
其关系为:

$$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3)$$

指数时间的关系为:

$$O(2^n) < O(n!) < O(n^n)$$

当n取得很大时, 指数时间算法和多项式时间算法在所需时间上非常悬殊。因此, 只要有人能将现有指数时间算法中的任何一个算法化简为多项式时间算法, 那就取得了一个伟大的成就。



## $O(n^k)$

$O(n^k)$ :  $k \geq 2$ ,  $k$ 次方时间阶

**例 1** 两个 $n$ 阶方阵的乘法

```
for(i=1, i<=n; ++i)
    for(j=1; j<=n; ++j)
        { c[i][j]=0;
          for(k=1; k<=n; ++k)
              c[i][j]+=a[i][k]*b[k][j]; }
```

由于是一个三重循环, 每个循环从1到 $n$ , 则总次数为:  $n \times n \times n = n^3$  时间复杂度为  $T(n) = O(n^3)$

**例 2** {++x; s=0;}

将 $x$ 自增看成是基本操作, 则语句频度为 1, 即时间复杂度为  $O(1)$ 。

如果将 $s=0$ 也看成是基本操作, 则语句频度为 2, 其时间复杂度仍为  $O(1)$ , 即常量阶。

## $O(n^k)$

**例 3** for(i=1; i<=n; ++i)  
    { ++x; s+=x; }

语句频度为:  $2n$ , 其时间复杂度为:  $O(n)$ , 即为线性阶。

**例 4** for(i=1; i<=n; ++i)  
    for(j=1; j<=n; ++j)  
        { ++x; s+=x; }

语句频度为:  $2n^2$ , 其时间复杂度为:  $O(n^2)$ , 即为平方阶。



## $O(n^k)$

**定理：**若 $A(n) = a_m n^m + a_{m-1} n^{m-1} + \dots + a_1 n + a_0$ 是一个 $m$ 次多项式，则 $A(n) = O(n^m)$

**例 5**

```
for(i=2; i<=n; ++i)
    for(j=2; j<=i-1; ++j)
        { ++x; a[i,j]=x; }
```

语句频度为： $1 + 2 + 3 + \dots + n - 2 = (1 + n - 2) \times (n - 2) / 2$   
 $= (n - 1)(n - 2) / 2 = n^2 - 3n + 2$

$\therefore$  时间复杂度为 $O(n^2)$ ，即此算法的时间复杂度为平方阶。

一个算法时间为 $O(1)$ 的算法，它的基本运算执行的次数是固定的。因此，总的时间由一个常数（即零次多项式）来限界。而一个时间为 $O(n^2)$ 的算法则由一个二次多项式来限界。

**有的情况下，算法中基本操作重复执行的次数还随问题的输入数据集不同而不同。**

**例1：素数的判断算法。**

```
void prime( int n)
/* n是一个正整数 */
{ int i=2;
  while ( (n%i)!=0 && i*1.0< sqrt(n) ) i++;
  if (i*1.0>sqrt(n) )
    printf( "&d 是一个素数\n" , n);
  else
    printf( "&d 不是一个素数\n" , n);
}
```

嵌套的最深层语句是 $i++$ ；其频度由条件 $(n \% i) != 0 \ \&\& \ i * 1.0 < \sqrt{n}$ 决定，显然 $i * 1.0 < \sqrt{n}$ ，时间复杂度 $O(n^{1/2})$ 。

例2：冒泡排序法。

```
void bubble_sort(int a[], int n)
{
    change=false;
    for (i=n-1; change=TURE; i>1 && change; --i)
        for (j=0; j<i; ++j)
            if (a[j]>a[j+1])
                { a[j] ↔ a[j+1]; change=TURE; }
}
```

- 最好情况：0次
- 最坏情况： $1 + 2 + 3 + \dots + n - 1 = n(n - 1)/2$
- 平均时间复杂度为： $O(n^2)$

51

51

## 对数阶 $O(\log n)$ 与线性对数 $O(n \log n)$

$O(\log n)$ :

```
int i = 1;
while(i < n)
{
    i = i * 2;
}
```

$O(n \log n)$ :

```
for (m = 1; m < n; m++)
{
    i = 1;
    while (i < n)
    {
        i = i * 2;
    }
}
```

52

52

## 算法的空间分析

**空间复杂度(Space complexity)**：是指算法编写成程序后，在计算机中运行时所需存储空间大小的度量。

记作： $S(n) = O(f(n))$

其中：n为问题的规模(或大小)

该存储空间一般包括三个方面：

1. 指令常数变量所占用的存储空间;
2. 输入数据所占用的存储空间;
3. **辅助(存储)空间。**

一般地，算法的**空间复杂度**指的是**辅助空间**。

- 一维数组a[n]：空间复杂度  $O(n)$
- 二维数组a[n][m]：空间复杂度  $O(n * m)$

## 算法的存储空间需求

- 若输入数据所占空间只取决于问题本身，和算法无关，则**只需要分析除输入和程序之外的辅助变量所占额外空间**。
- 若所需额外空间相对于输入数据量来说是常数，则称此算法为**原地工作**。
- 若所需存储量依赖于特定的输入，则**通常按最坏情况考虑**。

专业铸就品质 梦想成就未来

The Specialty Casts Quality,the Dream Gains Future.

### 第三章 常见算法思想及案例

## 常见算法思想

0. 穷举法
1. 递归与分治策略
2. 动态规划法
3. 贪心法
4. 回溯法
5. 分支限界法
6. 概率算法

## 0-穷举法

### 穷举法 的基本概念

- 穷举法也称为**枚举法**(Exhaustive Attack method), 或称**暴力破解法**, 又称为**强力法**(Brute-force method), **完全试凑法**(complete trial-and-error method)。它的基本思想是不重复、不遗漏地穷举所有可能情况, 或把信息条理化、系统化、或进行分类, 寻找规律, 引出信息, 以便从中寻找满足条件的结果。
- 穷举法常用于解决“**是否存在**”、“**有多少种情况**”等类型的问题。对于一些**数学问题**、**逻辑推理**问题, 穷举法看来也是一种“笨”方法, 但它恰好利用了计算机高速运算的特点, 可以避免复杂的逻辑推理过程, 使问题简单化。

## 运用实例

- 求1~100的素数
- 计算1~100一共多少个9
- 求所有的水仙花数的个数
- 求10000以内的完数
- 马克思手稿中有一道趣味数学题：有30个人，其中有男人、女人和小孩，在一家饭馆里吃饭共花了50先令，每个男人各花3先令，每个女人各花2先令，每个小孩各花1先令，问男人、女人和小孩各有几人？

$$\begin{cases} x + y + z = 30 \\ 3x + 2y + z = 50 \end{cases}$$

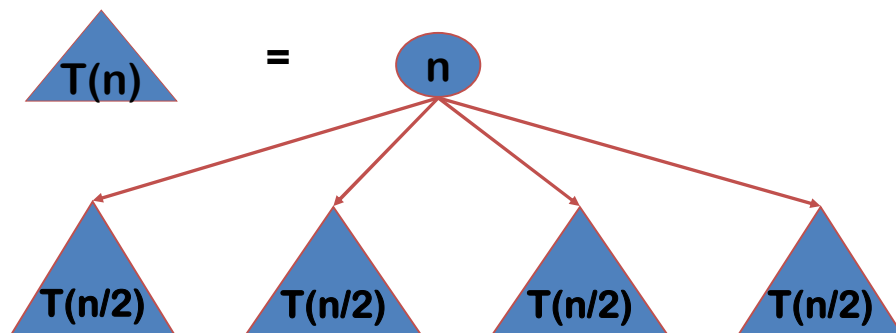
59

59

## 1-递归与分治策略、算法总体思想

将要求解的较大规模的问题分割成k个更小规模的子问题。

对这k个子问题分别求解。如果子问题的规模仍然不够小，则再划分为k个子问题，如此递归的进行下去，直到问题规模足够小，很容易求出其解为止。

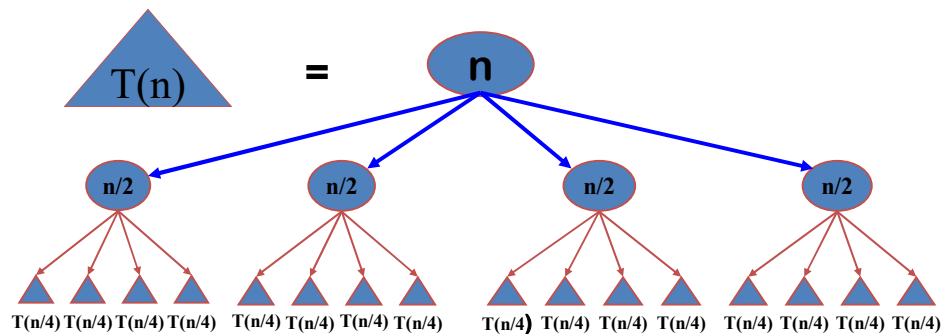


60

60

## 算法总体思想

- 对这 $k$ 个子问题分别求解。如果子问题的规模仍然不够小，则再划分为 $k$ 个子问题，如此递归的进行下去，直到问题规模足够小，很容易求出其解为止。

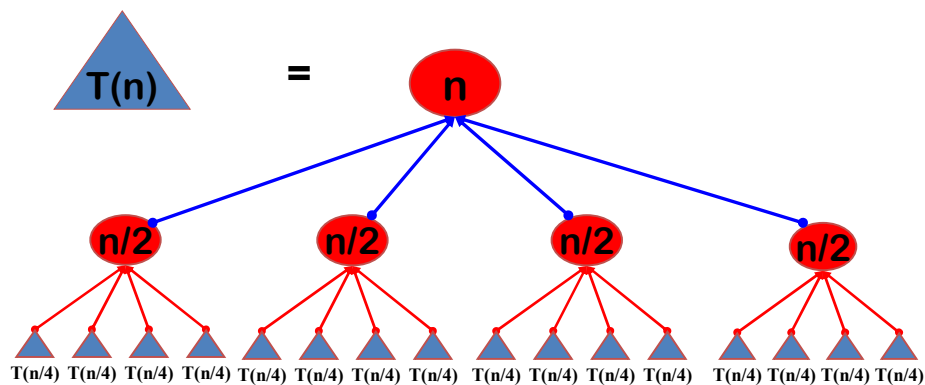


61

61

## 算法总体思想

将求出的小规模的问题的解合并为一个更大规模的问题的解，**自底向上**逐步求出原来问题的解。

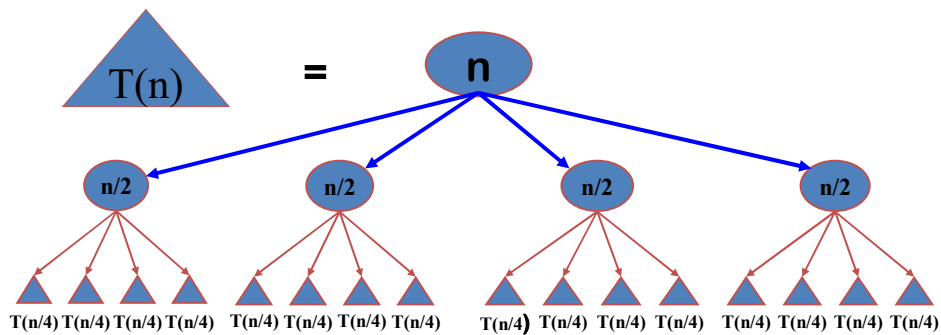


62

62

## 算法总体思想

对这 $k$ 个子问题分别求解。如果子问题的规模仍然不够小，则再划分为 $k$ 个子问题，如此递归的进行下去，直到问题规模足够小，很容易求出其解为止。

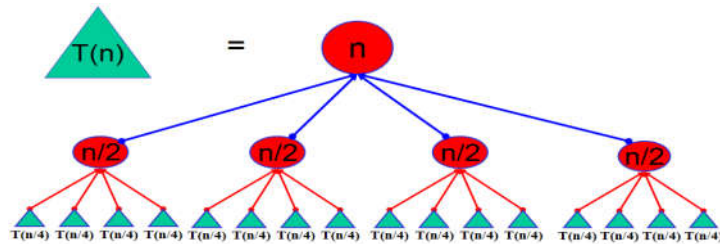


63

63

## 算法总体思想

将求出的小规模的问题的解合并为一个更大规模的问题的解，自底向上逐步求出原来问题的解。



分治法的设计思想是:将一个难以直接解决的大问题,分割成一些规模较小的相同问题,以便各个击破,分而治之。

凡治众如治寡,分数是也。

——孙子兵法

64

64



## 递归的概念

直接或间接地调用自身的算法称为**递归算法**。用函数自身给出定义的函数称为**递归函数**。

**由分治法产生的子问题往往是原问题的较小模式，这就为使用递归技术提供了方便。**在这种情况下，反复应用分治手段，可以使子问题与原问题类型一致而其规模却不断缩小，最终使子问题缩小到很容易直接求出其解。这自然导致递归过程的产生。

**分治与递归像一对孪生兄弟**，经常同时应用在算法设计之中，并由此产生许多高效算法。

65

65

## 例1 阶乘函数

阶乘函数可递归地定义为：

$$n! = \begin{cases} 1 & n = 0 \\ n(n-1)! & n > 0 \end{cases}$$

边界条件

递归方程

边界条件与递归方程是递归函数的二个要素，递归函数只有具备了这两个要素，才能在有限次计算后得出结果。

66

66

## 例2 Fibonacci数列

无穷数列1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ..., 被称为 Fibonacci数列。它可以递归地定义为:

$$F(n) = \begin{cases} 1 & n = 0 \\ 1 & n = 1 \\ F(n-1) + F(n-2) & n > 1 \end{cases}$$

边界条件

递归方程

```
int fib(int n)
{
    if (n == 0 || n == 1)
    {
        return n;
    }

    return fib(n - 1) + fib(n - 2);
}
```

67

联航精英训练营

67

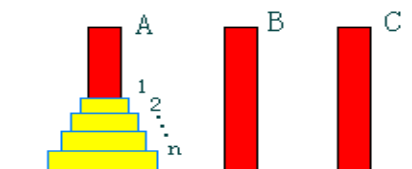
## 例3 Hanoi塔问题

设a,b,c是3个塔座。开始时, 在塔座a上有一叠共n个圆盘, 这些圆盘自下而上, 由大到小地叠在一起。各圆盘从小到大编号为1,2,...,n,现要求将塔座a上的这一叠圆盘移到塔座b上, 并按同样顺序叠置。在移动圆盘时应遵守以下移动规则:

**规则1:**每次只能移动1个圆盘;

**规则2:**任何时刻都不允许将较大的圆盘压在较小的圆盘之上;

**规则3:**在满足移动规则1和2的前提下,可将圆盘移至a,b,c中任一塔座上。



68

联航精英训练营

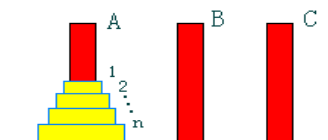
68

### 例3 Hanoi塔问题

在问题规模较大时，较难找到一般的方法，因此我们尝试用递归技术来解决这个问题。当 $n=1$ 时，问题比较简单。此时，只要将编号为1的圆盘从塔座a直接移至塔座b上即可。

当 $n > 1$ 时，需要利用塔座c作为辅助塔座。此时①若能设法将 $n-1$ 个较小的圆盘依照移动规则从塔座a移至塔座c，②将剩下的最大圆盘从塔座a移至塔座b，③再设法将 $n-1$ 个较小的圆盘依照移动规则从塔座c移至塔座b。

由此可见， $n$ 个圆盘的移动问题可分为2次 $n-1$ 个圆盘的移动问题，这又可以递归地用上述方法来做。由此可以设计出解Hanoi塔问题的递归算法。



69

### 递归小结

- **优点：**结构清晰，可读性强，而且容易用数学归纳法来证明算法的正确性，因此它为设计算法、调试程序带来很大方便。
- **缺点：**递归算法的运行效率较低，无论是耗费的计算时间还是占用的存储空间都比非递归算法要多。

70

## 分治法的适用条件

分治法所能解决的问题一般具有以下几个特征：

1. 该问题的规模缩小到一定的程度就可以容易地解决；
2. 该问题可以分解为若干个规模较小的相同问题，即该问题具有**最优子结构性质**；
3. 利用该问题分解出的子问题的解可以合并为该问题的解；
4. 该问题所分解出的各个子问题是相互独立的，即子问题之间不包含公共的子问题。

1. 因为问题的计算复杂性一般是随着问题规模的增加而增加，因此大部分问题满足这个特征。

2. 这条特征是应用分治法的前提，它也是大多数问题可以满足的，此特征反映了递归思想的应用

3. 能否利用分治法完全取决于问题是否具有这条特征，如果具备了前两条特征，而不具备第三条特征，则可以考虑**贪心算法**或**动态规划**。

4. 这条特征涉及到分治法的效率，如果各子问题是不独立的，则分治法要做许多不必要的工作，重复地解公共的子问题，此时虽然也可用分治法，但一般用**动态规划**较好。

## 分治法的基本步骤

```
divide-and-conquer(P)
{
    if ( | P | <= n0) adhoc(P); //解决小规模的问题
    divide P into smaller subinstances P1, P2, ..., Pk; //分解问题
    for (i=1, i<=k, i++)
        yi = divide-and-conquer(Pi); //递归的解各子问题
    return merge(y1, ..., yk); //将各子问题的解合并为原问题的解
}
```

人们从大量实践中发现，在用分治法设计算法时，**最好使子问题的规模大致相同**。即将一个问题分成大小相等的k个子问题的处理方法是行之有效的。这种使子问题规模大致相等的做法是出自一种**平衡(balancing)子问题**的思想，它几乎总是比子问题规模不等的做法要好。

## 分治法的复杂性分析

一个分治法将规模为 $n$ 的问题分成 $k$ 个规模为 $n/m$ 的子问题去解。设分解阈值 $n_0=1$ ，且 $\text{ad hoc}$ 解规模为1的问题耗费1个单位时间。再设将原问题分解为 $k$ 个子问题以及用merge将 $k$ 个子问题的解合并为原问题的解需用 $f(n)$ 个单位时间。用 $T(n)$ 表示该分治法解规模为 $|P|=n$ 的问题所需的计算时间，则有：

$$T(n) = \begin{cases} O(1) & n = 1 \\ kT(n/m) + f(n) & n > 1 \end{cases}$$

通过迭代法求得方程的解： $T(n) = n^{\log_m k} + \sum_{j=0}^{\log_m n - 1} k^j f(n/m^j)$

**注意：**递归方程及其解只给出 $n$ 等于 $m$ 的方幂时 $T(n)$ 的值，但是如果认为 $T(n)$ 足够平滑，那么由 $n$ 等于 $m$ 的方幂时 $T(n)$ 的值可以估计 $T(n)$ 的增长速度。通常假定 $T(n)$ 是单调上升的，从而当 $m^i \leq n < m^{i+1}$ 时， $T(m^i) \leq T(n) < T(m^{i+1})$ 。

## 例4 二分搜索技术

给定已按升序排好序的 $n$ 个元素 $a[0:n-1]$ ，现要在这 $n$ 个元素中找出一特定元素 $x$ 。

**分析1：**如果 $n=1$ 即只有一个元素，则只要比较这个元素和 $x$ 就可以确定 $x$ 是否在表中。因此这个问题满足分治法的第一个适用条件

**分析2：**

- 该问题的规模缩小到一定的程度就可以容易地解决；
- 该问题可以分解为若干个规模较小的相同问题；
- 分解出的子问题的解可以合并为原问题的解；
- 分解出的各个子问题是相互独立的。

**分析3：**比较 $x$ 和 $a$ 的中间元素 $a[\text{mid}]$ ，若 $x=a[\text{mid}]$ ，则 $x$ 在 $L$ 中的位置就是 $\text{mid}$ ；如果 $x < a[\text{mid}]$ ，由于 $a$ 是递增排序的，因此假如 $x$ 在 $a$ 中的话， $x$ 必然排在 $a[\text{mid}]$ 的前面，所以我们只要在 $a[\text{mid}]$ 的前面查找 $x$ 即可；如果 $x > a[\text{mid}]$ ，同理我们只要在 $a[\text{mid}]$ 的后面查找 $x$ 即可。无论是在前面还是后面查找 $x$ ，其方法都和在 $a$ 中查找 $x$ 一样，只不过是查找的规模缩小了。这就说明了此问题满足分治法的第二个和第三个适用条件。

**分析4：**很显然此问题分解出的子问题相互独立，即在 $a[i]$ 的前面或后面查找 $x$ 是独立的子问题，因此满足分治法的第四个适用条件。

## 二分搜索技术

给定已按升序排好序的n个元素a[0:n-1]，现要在这n个元素中找出一特定元素x。

据此容易设计出**二分搜索算法**：

```
int binarySearch(int [] a, int x, int n)
{
    // 在 a[0] <= a[1] <= ... <= a[n-1] 中搜索 x
    // 找到x时返回其在数组中的位置，否则返回-1
    int left = 0; int right = n - 1;
    while (left <= right) {
        int middle = (left + right)/2;
        if (x == a[middle]) return middle;
        if (x > a[middle]) left = middle + 1;
        else right = middle - 1;
    }
    return -1; // 未找到x
}
```

### 算法复杂度分析：

每执行一次算法的while循环，待搜索数组的大小减少一半。因此，在最坏情况下，while循环被执行了O(logn)次。循环体内运算需要O(1)时间，因此整个算法在最坏情况下的计算时间复杂度为O(logn)。

## 例5归并排序

**基本思想：**将待排序元素分成大小大致相同的2个子集合，分别对2个子集合进行排序，最终将排好序的子集合合并成为所要求的排好序的集合。

```
void mergeSort(Comparable a[], int left, int right)
{
    if (left < right) { //至少有2个元素
        int i = (left + right)/2; //取中点
        mergeSort(a, left, i);
        mergeSort(a, i + 1, right);
        merge(a, b, left, i, right); //合并到数组b
        copy(a, b, left, right); //复制回数组a
    }
}
```

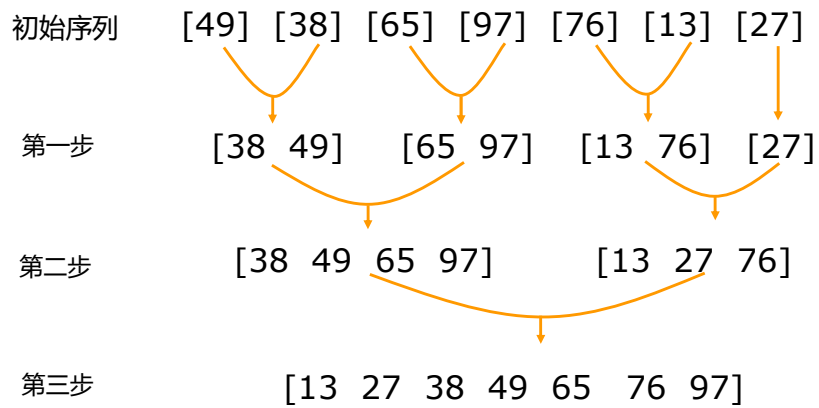
### 复杂度分析

$$T(n) = \begin{cases} O(1) & n \leq 1 \\ 2T(n/2) + O(n) & n > 1 \end{cases}$$

$T(n) = O(n \log n)$  渐进意义下的最优算法

## 归并排序

算法mergeSort的递归过程可以消去。



77

77

## 归并排序

最坏时间复杂度:  $O(n \log n)$

平均时间复杂度:  $O(n \log n)$

辅助空间:  $O(n)$

稳定性: 稳定

78

78

## 分治与二分法

- 利用分治策略求解时，所需时间取决于分解后子问题的个数、子问题的规模大小等因素，而二分法，由于其划分的简单和均匀的特点，是经常采用的一种有效的方法，例如二分法检索。

分治，分而治之。分的原因是因为问题的规模太大，需要拆开了解决，目的是为了解决问题，分解只是手段。所以分治的步骤其实很明确：

- **分解**：将大问题的分解成小问题，是这个算法的核心。也是使用分治法的效率保证，如果分解不合理。那么反而会弄巧成拙。
- **解决**：解决问题，便是分解之后的小问题。他们的解决步骤是相同，至少是相似的。所以，分治法中经常用到递归，就是基于这样的目的。
- **合并**：前面那么麻烦的两步，最终的目的仍然是为了解决这个问题。所以需要将分解问题得到的解，合并成最终需要的终极答案，便是这个算法的结束过程。譬如，你使用递归的时候，也需要最后退出的条件。分治法结束条件，就是合并步骤的结束。

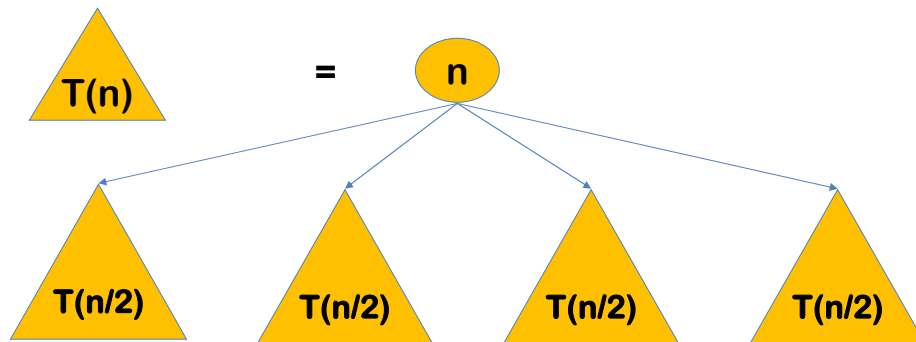
## 运用实例

- 汉诺塔
- Fibonacci数列
- 求n!
- 二分法查找
- 归并排序
- 快速排序
- 二叉树遍历



## 2. 动态规划 总体思想

- 动态规划算法与分治法类似，其基本思想也是将待求解问题分解成若干个子问题

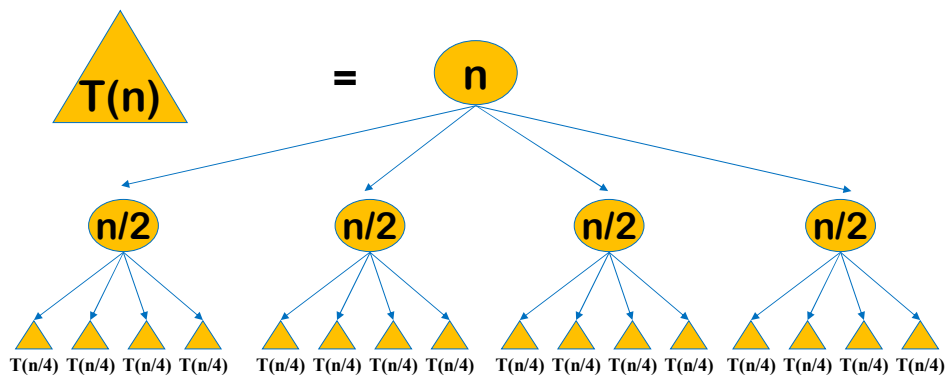


81

81

## 算法总体思想

- 但是经分解得到的子问题往往**不是互相独立的**。不同子问题的数目常常只有多项式量级。在用分治法求解时，有些子问题被重复计算了许多次。



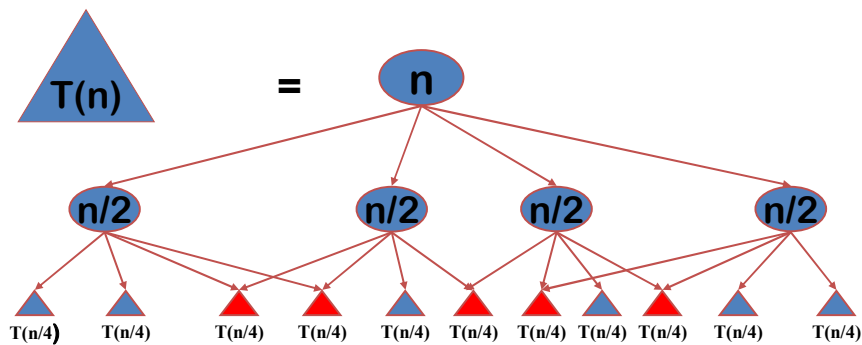
82

82

## 算法总体思想

如果能够**保存已解决的子问题的答案**，而在需要时再找出已求得的答案，就可以避免大量重复计算，从而得到多项式时间算法。

为了达到这个目的，可以用一个表来记录所有已经解决的子问题的答案，不管该子问题以后是否被用到，只要它被计算过，就将其结果填入表中。



83



联航精英训练营

83

## 动态规划法

**Those who cannot remember the past are doomed to repeat it.**

-----George Santayana,  
The life of Reason,  
Book I: Introduction and  
Reason in Common  
Sense (1905)

桑塔亚那的最著名的一句话为“那些不能铭记过去的人注定要重蹈覆辙。” 《社会中的理性》

84



联航精英训练营

84

## 动态规划基本步骤

1. 找出**最优解**的性质，并刻画其结构特征。
2. 递归地定义最优值。
3. 以**自底向上**的方式计算出最优值。
4. 根据计算最优值时得到的信息，构造最优解。

85



联航精英训练营

85

## 运用案例

- 斐波拉契数列
- 台阶问题
- 0-1背包问题

86



联航精英训练营

86

## 问题：台阶问题

有 $n$ 级台阶，一个人每次上一级或者两级，问有多少种走完 $n$ 级台阶的方法。

**分析：**动态规划的实现的关键在于能不能准确合理的用动态规划表来抽象出实际问题。

在这个问题上，我们让 $f(n)$ 表示走上 $n$ 级台阶的方法数。

那么当 $n$ 为1时， $f(n) = 1$ ， $n$ 为2时， $f(n) = 2$ ，就是说当台阶只有一级的时候，方法数是一种，台阶有两级的时候，方法数为2。

那么当我们要走上 $n$ 级台阶，必然是从 $n-1$ 级台阶迈一步或者是从 $n-2$ 级台阶迈两步，所以到达 $n$ 级台阶的方法数必然是到达 $n-1$ 级台阶的方法数加上到达 $n-2$ 级台阶的方法数之和，即 $f(n) = f(n-1) + f(n-2)$ ，我们用 $dp[n]$ 来表示动态规划表， $dp[i], i > 0 \ \&\& \ i \leq n$ ，表示到达 $i$ 级台阶的方法数。

## 题目：0-1背包问题

### 问题描述：

给定 $n$ 种物品和一背包。物品 $i$ 的重量是 $w_i$ ，其价值为 $v_i$ ，背包的容量为 $C$ 。问应如何选择装入背包的物品，使得装入背包中物品的总价值最大？

对于一种物品，要么装入背包，要么不装。所以对于一种物品的装入状态可以取0和1。我们设物品 $i$ 的装入状态为 $x_i$ ， $x_i \in (0, 1)$ ，此问题称为**0-1背包问题**。

### 数据：

物品个数 $n = 5$ ,

物品重量 $w[n] = \{2, 2, 6, 5, 4\}$ ,

物品价值 $v[n] = \{6, 3, 5, 4, 6\}$ ,

## 0-1 背包问题

0-1背包问题是一个特殊的整数规划问题。

$$\begin{aligned} \max \quad & \sum_{i=1}^n v_i x_i \\ \text{s.t.} \quad & \begin{cases} \sum_{i=1}^n w_i x_i \leq C \\ x_i \in \{0,1\}, 1 \leq i \leq n \end{cases} \end{aligned}$$

89



联航精英训练营

89

## 0-1 背包问题

设所给0-1背包问题的子问题

$$\max \sum_{k=i}^n v_k x_k \quad \begin{cases} \sum_{k=i}^n w_k x_k \leq j \\ x_k \in \{0,1\}, i \leq k \leq n \end{cases}$$

的最优值为 $m(i, j)$ ，即 $m(i, j)$ 是**背包容量为j**，可选择物品为 $i, i+1, \dots, n$ 时0-1背包问题的**最优值**。由0-1背包问题的最优子结构性质，可以建立计算 $m(i, j)$ 的递归式如下。

$$m(i, j) = \begin{cases} \max\{m(i-1, j), m(i-1, j-w_i) + v_i\} & j \geq w_i \\ m(i-1, j) & 0 \leq j < w_i \end{cases}$$

$$m(n, j) = \begin{cases} v_n & j \geq w_n \\ 0 & 0 \leq j < w_n \end{cases}$$

**算法复杂度分析：**

从 $m(i, j)$ 的递归式容易看出，算法需要 $O(nc)$ 计算时间。当背包容量 $c$ 很大时，算法需要的计算时间较多。例如，当 $c > 2^n$ 时，算法需要 $\Omega(n2^n)$ 计算时间。

90



联航精英训练营

90

### 3-贪心法

- 顾名思义，贪心算法总是作出在**当前看来最好的选择**。也就是说贪心算法并不从整体最优考虑，它所作出的选择只是在某种意义上的局部最优选择。
- 当然，**希望贪心算法得到的最终结果也是整体最优的**。虽然贪心算法不能对所有问题都得到整体最优解，但对许多问题它能产生整体最优解。如选择排序，最小生成树问题等。在一些情况下，即使贪心算法不能得到整体最优解，其最终结果却是最优解的很好近似。

### 贪心算法的基本要素

#### 1. 贪心选择性质

所谓**贪心选择性质**是指所求问题的**整体最优解**可以通过一系列**局部最优**的选择，即贪心选择来达到。这是贪心算法可行的第一个基本要素，也是贪心算法与动态规划算法的主要区别。

动态规划算法通常以**自底向上**的方式解各子问题，而贪心算法则通常以**自顶向下**的方式进行，以迭代的方式作出相继的贪心选择，每作一次贪心选择就将所求问题简化为规模更小的子问题。

对于一个具体问题，要确定它是否具有贪心选择性质，必须**证明每一步所作的贪心选择最终导致问题的整体最优解**。

## 贪心算法的基本要素

### 2. 最优子结构性质

当一个问题最优解包含其子问题的最优解时，称此问题具有**最优子结构性质**。

问题的最优子结构性质是该问题可用**动态规划算法**或**贪心算法**求解的关键特征。

## 贪心算法的基本要素

### 3. 贪心算法与动态规划算法的差异

贪心算法和动态规划算法都要求问题具有最优子结构性质，这是两类算法的一个共同点。

但是，对于具有**最优子结构**的问题应该选用贪心算法还是动态规划算法求解？

**是否能用动态规划算法求解的问题也能用贪心算法求解？**

下面研究2个经典的**组合优化问题**，并以此说明贪心算法与动态规划算法的主要差别。

## 贪心算法的基本要素

### 0-1背包问题:

#### • 问题描述:

给定 $n$ 种物品和一背包。物品 $i$ 的重量是 $w_i$ ，其价值为 $v_i$ ，背包的容量为 $C$ 。问应如何选择装入背包的物品，使得装入背包中物品的总价值最大？

- 对于一种物品，要么装入背包，要么不装。所以对于一种物品的装入状态可以取0和1。我们设物品 $i$ 的装入状态为 $x_i$ ， $x_i \in (0, 1)$ ，此问题称为**0-1背包问题**。

#### • 数据:

- 物品个数 $n = 5$ ,
- 物品重量 $w[n] = \{2, 2, 6, 5, 4\}$ ,
- 物品价值 $V[n] = \{6, 3, 5, 4, 6\}$ ,

95

## 贪心算法的基本要素

### ■ 背包问题:

与0-1背包问题类似，所不同的是在选择物品 $i$ 装入背包时，**可以选择物品 $i$ 的一部分**，而不一定要全部装入背包， $1 \leq i \leq n$ 。

这2类问题都具有**最优子结构**性质，极为相似，但背包问题可以用贪心算法求解，而0-1背包问题却不能用贪心算法求解。

96



## 贪心算法的基本要素

### 用贪心算法解背包问题的基本步骤：

首先计算每种物品单位重量的价值 $v_i/w_i$ ，

然后，依贪心选择策略，将尽可能多的**单位重量价值最高**的物品装入背包。

若将这种物品全部装入背包后，背包内的物品总重量未超过C，则选择单位重量价值次高的物品并尽可能多地装入背包。

依此策略一直地进行下去，直到背包装满为止。

## 贪心算法的基本要素

对于**0-1背包问题**，贪心选择之所以不能得到最优解是因为在这种情况下，它无法保证最终能将背包装满，部分闲置的背包空间使每公斤背包空间的价值降低了。

事实上，在考虑0-1背包问题时，应比较选择该物品和不选择该物品所导致的最终方案，然后再作出最好选择。由此就导出许多互相重叠的子问题。这正是该问题可用**动态规划算法**求解的另一重要特征。

实际上也是如此，**动态规划算法**的确可以有效地解0-1背包问题。

## 贪心法的基本思路：

1. 建立数学模型来描述问题。
2. 把求解的问题分成若干个子问题。
3. 对每一子问题求解，得到子问题的局部最优解。
4. 把子问题的解局部最优解合成原来解问题的一个解。

### 贪心算法适用的问题

贪心策略适用的前提是：**局部最优策略能导致产生全局最优解。**

贪心选择是指所求问题的整体最优解可以通过一系列局部最优的选择，即贪心选择来达到。这是贪心算法可行的第一个基本要素，也是贪心算法与动态规划算法的主要区别。贪心选择是采用**从顶向下**、以迭代的方法做出相继选择，**每做一次贪心选择就将所求问题简化为一个规模更小的子问题。**对于一个具体问题，要确定它是否具有贪心选择的性质，**我们必须证明每一步所作的贪心选择最终能得到问题的最优解。**通常可以首先证明问题的一个整体最优解，是从贪心选择开始的，而且作了贪心选择后，原问题简化为一个规模更小的类似子问题。然后，用数学归纳法证明，通过每一步贪心选择，最终可得到问题的一个整体最优解。

## 运行实例

冒泡排序

选择排序

背包问题

找零钱问题

活动安排问题

最优装载问题

## 4-回溯法 (Backtracking)

有许多问题，当需要找出它的解集或者要求回答什么解是满足某些约束条件的最佳解时，往往要使用回溯法。

回溯法的基本做法是搜索，或是一种组织得井井有条的，能避免不必要搜索的穷举式搜索法。这种方法适用于解一些组合数相当大的问题。

回溯法在问题的解空间树中，按深度优先策略，从根结点出发搜索解空间树。

算法搜索至解空间树的任意一点时，先判断该结点是否包含(而不是找到解)问题的解。如果肯定不包含，则跳过对该结点为根的子树的搜索(剪枝)，逐层向其祖先结点回溯；否则，进入该子树，继续按深度优先策略搜索。

## 回溯法的思想

回溯算法是所有搜索算法中最为基本的一种算法,是一种能避免不必要搜索的穷举式的搜索算法，其基本思想就是穷举搜索。

### 算法思想：

采用了一种“走不通就掉头”的思想。搜索时往往有多分支，按某一支为新的出发点，继续向下探索，当所有可能情况都探索过且都无法到达目标的时候，再回退到上一个出发点，继续探索另一个可能情况，这种不断回头寻找目标的方法称为“回溯法”。

## 回溯三要素

### 搜索的方式

主要采用**深度优先搜索**的方式

### 回溯三要素：

- 1) 解空间：该空间包含问题的解
- 2) 约束条件
- 3) 状态树

103

103

## 问题的解空间

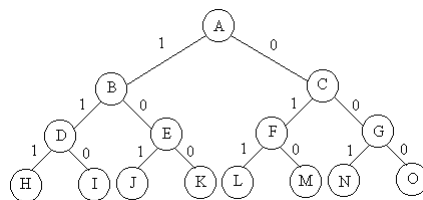
**问题的解向量：**回溯法希望一个问题的解能够表示成一个 $n$ 元式  $(x_1, x_2, \dots, x_n)$  的形式。

**显约束：**对分量 $x_i$ 的取值限定。

**隐约束：**为满足问题的解而对不同分量之间施加的约束。

**解空间：**对于问题的一个实例，解向量满足显式约束条件的**所有多元组**，构成

注意：同一个问题可以有多种表示，有些表示方法更简单，所需表示的状态空间更小（存储量少，搜索方法简单）。



$n=3$ 时的0-1背包问题用完全二叉树表示的解空间

104

104

## 回溯法的基本思想

- (1) 针对所给问题，定义问题的解空间；
- (2) 确定易于搜索的解空间结构；
- (3) 以深度优先方式搜索解空间，并在搜索过程中用剪枝函数避免无效搜索。

常用剪枝函数：

- ① 用**约束函数**在扩展结点处剪去不满足约束的子树；
- ② 用**限界函数**剪去得不到最优解的子树。

用回溯法解题的一个显著特征是在**搜索过程中动态产生问题的解空间**。在任何时刻，算法只保存从根结点到当前扩展结点的路径。如果解空间树中从根结点到叶结点的最长路径的长度为 $h(n)$ ，则回溯法所需的计算空间通常为 $O(h(n))$ 。而显式地存储整个解空间则需要 $O(2^{h(n)})$ 或 $O(h(n)!)$ 内存空间。

105



联航精英训练营

105

## 递归回溯

回溯法对解空间作**深度优先搜索**，因此，在一般情况下用递归方法实现回溯法。

```
void backtrack (int t)
{
    if (t>n)
        output(x);
    else
        for (int i=f(n,t); i<=g(n,t); i++)
        {
            x[t]=h(i);
            if (constraint(t)&&bound(t)) 约束和限定
                backtrack(t+1);
        }
}
```

106



联航精英训练营

106

## 迭代回溯

采用树的非递归深度优先遍历算法，可将回溯法表示为一个非递归迭代过程。

```
void iterativeBacktrack ()
{
    int t=1;
    while (t>0)
    {
        if (f(n,t)<=g(n,t))
            for (int i=f(n,t);i<=g(n,t);i++)
            {
                x[t]=h(i);
                if (constraint(t)&&bound(t))
                {
                    if (solution(t)) output(x);
                    else t++;
                }
            }
        else t--;
    }
}
```

107

107

## N皇后问题

在一个 $n \times n$ 的国际象棋棋盘上放置 $n$ 个皇后，使得它们中任意2个之间都不互相“攻击”，即任意2个皇后不可在同行、同列、同斜线上。

输出N,

- (1)求N皇后问题的一种放法;
- (2)求N皇后问题的所有放法

分析:

N=4时，右图是一组解

### 要素一：解空间

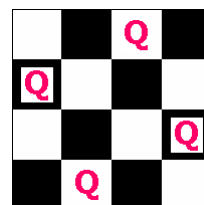
一般想法：利用二维数组，用 $[i,j]$ 确定一个皇后位置！

优化：利用约束条件，只需一维数组即可！

$x$ :array[1..n] of integer;

$x[i]$ : i表示第i行皇后

$x[i]$ 表示第i行上皇后放第几列



$x[3,1,4,2]$

108

108

# N皇后问题

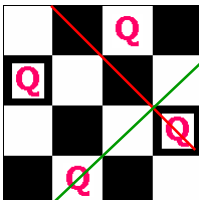
---

## 要素二：约束条件

**不同行：**数组x的下标保证不重复  
**不同列：** $x[i] \neq x[j] \quad (i \leq l, j \leq n; i < j)$   
**不同对角线：** $abs(x[i]-x[j]) \neq abs(i-j)$

填到第K行时，就与前1~(K-1)行都进行比较


```
Function Place(k:integer):boolean;  
    place:=true;  
    for j←1 to k-1 do  
        if |k-j|=|x[j]-x[k]| or x[j]=x[k] then  
            place:= false
```



## 要素三：状态树

将搜索过程中的每个状态用树的形式表示出来！  
画出状态树对书写程序有很大帮助！

109

 联航精英训练营



联航精英训练营

**K=0**

**过程：**进入新一行，该行上按顺序逐个格子尝试，直到能放为止（不冲突、不越界）

**K=1**

**算法描述：**

1. 产生一种新放法
2. 冲突，继续找，直到找到不冲突----不超范围
3. if 不冲突 then  $k < n \rightarrow k+1$   
 $k = n \rightarrow$ 一组解
4. if 冲突 then 回溯

**K=2**

**K=3**

**K=4**

回溯

回溯

回溯

出解后可以继续刚才的做法

110

## N皇后问题

### 程序结束条件:

一组解: 设标志, 找到一解后更改标志, 以标志做为结束循环的条件  
所有解:  $k=0$

### 判断约束函数:

```
Function Place(k:integer):boolean;  
  place:=true;  
  for j←1 to k-1 do  
    if  $|k-j|=|x[j]-x[k]|$  or  $x[j]=x[k]$  then  
      place:= false
```

### 回溯部份:

即状态恢复, 使其恢复到进入该分支前的状态, 继续新的分支  
 $x[k]:=0$ ;  
Dec(k);

### 程序实现:

回溯算法可用非递归和递归两种方法实现!

111



111

## 0-1背包问题

### 问题描述:

给定 $n$ 种物品和一背包。物品 $i$ 的重量是 $w_i$ , 其价值为 $v_i$ , 背包的容量为 $C$ 。问应如何选择装入背包的物品, 使得装入背包中物品的总价值最大?

对于一种物品, 要么装入背包, 要么不装。所以对于一种物品的装入状态可以取0和1。我们设物品 $i$ 的装入状态为 $x_i$ ,  $x_i \in (0, 1)$ , 此问题称为**0-1背包问题**。

### 数据:

物品个数 $n = 5$ ,

物品重量 $w[n] = \{2, 2, 6, 5, 4\}$ ,

物品价值 $V[n] = \{6, 3, 5, 4, 6\}$ ,

112



112



## 5-分支限界法的基本思想

### 1. 分支限界法与回溯法的不同

- (1) **求解目标不同**：回溯法的求解目标是找出解空间树中满足约束条件的所有解，而**分支限界法的求解目标则是找出满足约束条件的一个解**，或是在满足约束条件的解中找出在某种意义下的最优解。
- (2) **搜索方式不同**：回溯法以**深度优先**的方式搜索解空间树，而分支限界法则以**广度优先**或以**最小耗费优先**的方式搜索解空间树。

## 分支限界法的基本思想

### 2. 分支限界法基本思想

分支限界法常以**广度优先**或以**最小耗费（最大效益）优先**的方式搜索问题的解空间树。

在分支限界法中，每一个活结点只有一次机会成为扩展结点。活结点一旦成为扩展结点，就一次性产生其所有孩子结点。在这些孩子结点中，导致不可行解或导致非最优解的孩子结点被舍弃，其余孩子结点被加入活结点表中。

此后，从活结点表中取下一结点成为当前扩展结点，并重复上述结点扩展过程。这个过程一直持续到找到所需的解或活结点表为空时为止。

## 分支限界法的基本思想

### 3. 常见的两种分支限界法

#### (1) 队列式(FIFO)分支限界法

按照队列先进先出 (FIFO) 原则选取下一个节点为扩展节点。

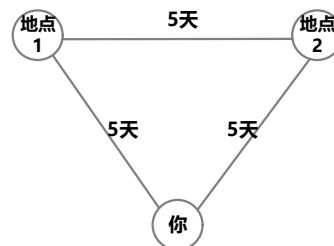
#### (2) 优先队列式分支限界法

按照优先队列中规定的优先级选取优先级最高的节点成为当前扩展节点。

## 6、概率算法

- 概率算法也叫**随机化**算法。概率算法允许算法在执行过程中随机地选择下一个计算步骤。在很多情况下，算法在执行过程中面临选择时，随机性选择比最优选择省时，因此概率算法可以在很大程度上降低算法的复杂度。
- 概率算法的一个基本特征是对所求解问题的同一实例用同一概率算法求解两次可能得到完全不同的效果。这两次求解问题所需的时间甚至所得到的结果可能会有相当大的差别。

## 概率算法-引言



**故事：**想象自己是神化故事的主人公，你有一张不易懂的地图，上面描述了一处宝藏的藏宝地点。

经分析你能确定最有可能的两个地点是藏宝地点，但二者相距甚远。假设你如果已到达其中一处，就立即知道该处是否为藏宝地点。你到达两处之一地点及从其中一处到另一处的距离是5天的行程。

进一步假设有一条恶龙，每晚光顾宝藏并从中拿走一部分财宝。

117

117

## 概率算法-引言

假设你取宝藏的方案有两种：

**方案1.** 花4天的时间计算出准确的藏宝地点，然后出发寻宝，一旦出发不能重新计算

**方案2.** 有一个小精灵告诉你地图的秘密，但你必须付给他报酬，相当于龙3晚上拿走的财宝

Prob: 若忽略可能的冒险和出发寻宝的代价，你是否接受小精灵的帮助？

显然，应该接受小精灵的帮助，因为你只需给出3晚上被盗窃的财宝量，否则你将失去4晚被盗财宝量。

但是，若冒险，你可能做得更好！

118

118

## 概率算法-引言

设 $x$ 是你决定之前当日的宝藏价值，设 $y$ 是恶龙每晚盗走的宝藏价值，并设 $x > 9y$

**方案1:** 4天计算确定地址，行程5天，你得到的宝藏价值为： $x - 9y$

**方案2:** 3y付给精灵，行程5天失去5y，你得到的宝藏价值为： $x - 8y$

**方案3:** 投硬币决定先到一处，失败后到另一处(冒险方案)

一次成功所得： $x - 5y$ ，机会 $1/2$   
二次成功所得： $x - 10y$ ，机会 $1/2$  } 期望赢利： $x - 7.5y$

## 意义

该故事告诉我们：**当一个算法面临某种选择时，有时随机选择比耗时做最优选择更好**，尤其是当最优选择所花的时间大于随机选择的平均时间的时候

显然，概率算法只能是期望的时间更有效，但它有可能遭受到最坏的可能性。

## 期望时间和平均时间的区别

### 确定算法的平均执行时间

输入规模一定的所有输入实例是等概率出现时，算法的平均执行时间。

### 概率算法的期望执行时间

反复解同一个输入实例所花的平均执行时间。

因此，对概率算法可以讨论如下两种期望时间

- ① **平均的期望时间**：所有输入实例上平均的期望执行时间
- ② **最坏的期望时间**：最坏的输入实例上的期望执行时间

## 数值概率算法

这类算法主要用于找到一个数字问题的近似解

### $\pi$ 值计算

- 实验：将  $n$  根飞镖随机投向一正方形的靶子，计算落入此正方形的内切圆中的飞镖数目  $k$ 。

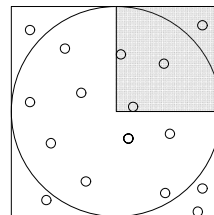
假定飞镖击中方靶子任一点的概率相等(用计算机模拟比任一飞镖高手更能保证此假设成立)

设圆的半径为  $r$ ，面积  $s_1 = \pi r^2$ ；方靶面积  $s_2 = 4r^2$

由等概率假设可知落入圆中的飞镖和正方形内的飞镖平均比为：

$$\text{由此知：} \quad \frac{k}{n} = \frac{\pi r^2}{4r^2} = \frac{\pi}{4}$$

$$\pi \approx 4k / n \quad k = \frac{\pi n}{4}$$



## π 值计算

### 求π近似值的算法

为简单起见，只以上图的右上1/4象限为样本

```
Darts (n) {  
    k ← 0;  
    for i ← 1 to n do {  
        x ← uniform(0, 1);  
        y ← uniform(0, 1); // 随机产生点(x,y)  
        if (x2 + y2 ≤ 1) then k++; // 圆内  
    }  
    return 4k/n;  
}
```

实验结果：π=3.141592654

n = 1000万: 3.140740, 3.142568 (2位精确)

n = 1亿: 3.141691, 3.141363 (3位精确)

n = 10亿: 3.141527, 3.141507 (4位精确)



专业铸就品质 梦想成就未来

The Specialty Casts Quality, the Dream Gains Future.

## 第四章 线性表

125

125

### 线性表

线性结构是**最常用、最简单**的一种数据结构。而线性表是一种典型的线性结构。其基本特点是线性表中的数据元素是有序且是有限的。在这种结构中：

- ① 存在一个唯一的被称为“第一个”的数据元素；
- ② 存在一个唯一的被称为“最后一个”的数据元素；
- ③ 除第一个元素外，每个元素均有唯一——一个直接前驱；
- ④ 除最后一个元素外，每个元素均有唯一——一个直接后继。

126

126

## 线性表的定义

### 线性表(Linear List) :

是由 $n(n \geq 0)$ 个数据元素(结点) $a_1, a_2, \dots, a_n$ 组成的有限序列。该序列中的所有结点具有相同的数据类型。其中数据元素的个数 $n$ 称为线性表的长度。

当 $n = 0$ 时, 称为空表。

当 $n > 0$ 时, 将非空的线性表记作:  $(a_1, a_2, \dots, a_n)$

$a_1$ 称为线性表的第一个(首)结点,  $a_n$ 称为线性表的最后一个(尾)结点。

$a_1, a_2, \dots, a_{i-1}$ 都是 $a_i(2 \leq i \leq n)$ 的**前驱**, 其中 $a_{i-1}$ 是 $a_i$ 的**直接前驱**;

$a_{i+1}, a_{i+2}, \dots, a_n$ 都是 $a_i(1 \leq i \leq n-1)$ 的**后继**, 其中 $a_{i+1}$ 是 $a_i$ 的**直接后继**。

## 线性表的逻辑结构

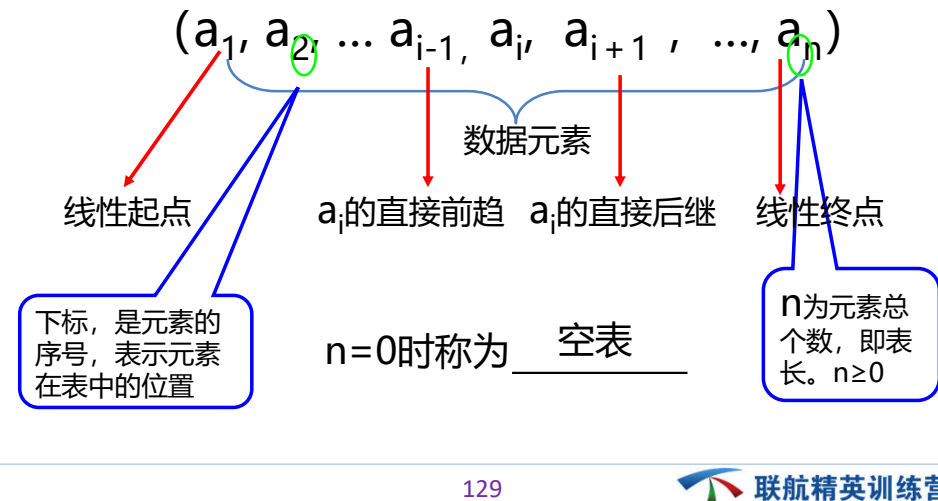
线性表中的数据元素 $a_i$ 所代表的具体含义随具体应用的不同而不同, 在线性表的定义中, 只不过是一个抽象的表示符号。

1. 线性表中的**结点**可以是**单值元素**(每个元素只有一个数据项)。
2. 线性表中的**结点**可以是**记录型**元素, 每个元素含有多个数据项, 每个项称为结点的一个域。每个元素有一个可以唯一标识每个结点的**数据项组**, 称为**关键字**。
3. 若线性表中的结点是**按值**(或按关键字值)由小到大(或由大到小)**排列**的, 称线性表是**有序的**。
4. 线性表是一种相当灵活的数据结构, 其长度可根据需要**增长**或**缩短**。
5. 对线性表的数据元素可以访问、插入和删除。



## 线性表的逻辑结构

**线性表的定义：**用数据元素的有限序列表示



129

例1 分析26个英文字母组成的英文表是什么结构。

(A, B, C, D, …… , Z)

分析：数据元素都是同类型（字母），元素间关系是线性的。

例2 分析学生情况登记表是什么结构。

学号	姓名	性别	年龄	班级
1406010402	陈杰			2014级计软14-1班
1406010405	邓博			2014级计软14-1班
1406010406	管杰			2014级计软14-1班
1406010410	黄腾达			2014级计软14-1班
1406010413	李荣智			2014级计软14-1班
:	:	:	:	:

分析：数据元素都是同类型（记录），元素间关系是线性的。

注意：同一线性表中的元素必定具有相同特性！

130

130

## 线性表的抽象数据类型定义

```
ADT List{
    数据对象:  $D = \{ a_i \mid a_i \in \text{ElemSet}, i=1,2,\dots,n, n \geq 0 \}$ 
    数据关系:  $R = \{ \langle a_{i-1}, a_i \rangle \mid a_{i-1}, a_i \in D, i=2,3,\dots,n \}$ 
    基本操作:
        InitList( &L )
        操作结果: 构造一个空的线性表L;
        ListLength( L )
        初始条件: 线性表L已存在;
        操作结果: 若L为空表, 则返回TRUE, 否则返回FALSE;
        ....
        GetElem( L, i, &e )
        初始条件: 线性表L已存在,  $1 \leq i \leq \text{ListLength}(L)$ ;
        操作结果: 用e返回L中第i个数据元素的值;
        ListInsert( L, i, &e )
        初始条件: 线性表L已存在,  $1 \leq i \leq \text{ListLength}(L)$ ;
        操作结果: 在线性表L中的第i个位置插入元素e;
        ...
    } ADT List
```

131



联航精英训练营

131

## 顺序表的表示

用一组地址连续的存储单元依次存储线性表的元素。

线性表的顺序表示又称为**顺序存储结构**或**顺序映像**。

顺序存储定义: 把逻辑上相邻的数据元素存储在物理上相邻的存储单元中的存储结构。

特点: **逻辑上相邻的元素, 物理上也相邻**

顺序存储方法:

可以利用数组V[n]来实现

注意: 在C语言中数组的下标是从0开始, 即:  
V[n]的有效范围是从 V[0] ~ V[n-1]

132



联航精英训练营

132

## 线性表的顺序存储结构

**顺序存储**：把线性表的结点按逻辑顺序依次存放在一组地址连续的存储单元里。用这种方法存储的线性表简称**顺序表**。

**链式存储**：用一组任意的存储单元存储线性表中的数据元素。用这种方法存储的线性表简称**线性链表**。

存储链表中结点的一组任意的存储单元可以是连续的，也可以是不连续的，甚至是零散分布在内存中的任意位置上的。

链表中结点的逻辑顺序和物理顺序不一定相同。

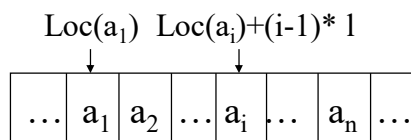
## 线性表的顺序存储结构

**顺序存储**：把线性表的结点按逻辑顺序依次存放在一组地址连续的存储单元里。用这种方法存储的线性表简称顺序表。

顺序存储的线性表的特点：

- ◆ 线性表的逻辑顺序与物理顺序一致；
- ◆ 数据元素之间的关系是以元素在计算机内“物理位置相邻”来体现。

设有非空的线性表：(a<sub>1</sub>, a<sub>2</sub>, ..., a<sub>n</sub>)。顺序存储如下图所示。



**在具体的机器环境下**：设线性表的每个元素需占用*l*个存储单元，以所占的第一个单元的存储地址作为数据元素的存储位置。则线性表中第*i*+1个数据元素的存储位置LOC(a<sub>i+1</sub>)和第*i*个数据元素的存储位置LOC(a<sub>i</sub>)之间满足下列关系：

$$\text{LOC}(a_{i+1}) = \text{LOC}(a_i) + l$$

线性表的第*i*个数据元素a<sub>i</sub>的存储位置为：

$$\text{LOC}(a_i) = \text{LOC}(a_1) + (i-1) * l$$

## C语言环境下顺序表的表示

在高级语言(如C语言)环境下：数组具有随机存取的特性，因此，借助数组来描述顺序表。除了用数组来存储线性表的元素之外，顺序表还应该有表示线性表的长度属性，所以用结构类型来定义顺序表类型。

```
#define OK 1
#define ERROR -1
#define MAX_SIZE 100

typedef int Status;
typedef int ElemType;
typedef struct sqlist
{
    ElemType Elem_array[MAX_SIZE];
    int length;
} SqList;
```

135



联航精英训练营

135

## 1. 顺序表的基本操作

顺序存储结构中，很容易实现线性表的一些操作：**初始化、赋值、查找、修改、插入、删除、求长度等。**

以下将对几种主要的操作进行讨论。

### 1 顺序线性表初始化

```
Status Init_SqList( SqList *L )
{
    L->elem_array=( ElemType * )malloc(MAX_SIZE*sizeof( ElemType ) );
    if ( !L -> elem_array ) return ERROR ;
    else { L->length= 0 ; return OK ; }
}
```

**修改** 通过数组的下标便可访问某个特定元素并修改之。

核心语句: `V[i]=x;`

136



联航精英训练营

136

## 2 顺序线性表的插入

在线性表  $L = (a_1, \dots, a_{i-1}, a_i, a_{i+1}, \dots, a_n)$  中的第  $i$  ( $1 \leq i \leq n$ ) 个位置上插入一个新结点  $e$ , 使其成为线性表:

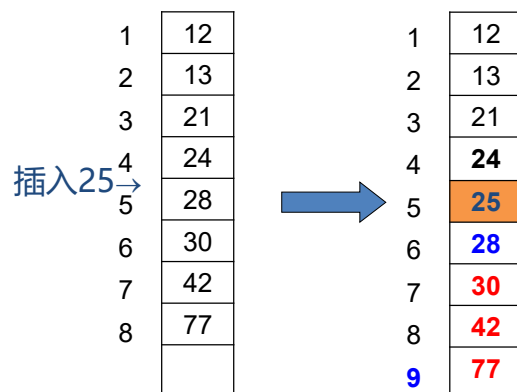
$$L = (a_1, \dots, a_{i-1}, e, a_i, a_{i+1}, \dots, a_n)$$

### 实现步骤

- (1) 将线性表  $L$  中的第  $i$  个至第  $n$  个结点后移一个位置。
- (2) 将结点  $e$  插入到结点  $a_{i-1}$  之后。
- (3) 线性表长度加1。

## 插入

在线性表的第  $i$  个位置前插入一个元素的示意图如下:



## 算法描述

```
Status Insert_SqList(SqList *L, int i, ElemType e)
{
    int j;
    if (i < 0 || i > L->length-1) return ERROR;
    if (L->length >= MAX_SIZE)
    { printf("线性表溢出!\n"); return ERROR; }

    /* i-1位置以后的所有结点后移 */
    for (j = L->length-1; j >= i-1; --j)
        L->Elem_array[j+1] = L->Elem_array[j];

    L->Elem_array[i-1] = e; /* 在i-1位置插入结点 */
    L->length++;
    return OK;
}
```

在顺序表上做插入运算，平均要移动表上一半结点。当表长 $n$ 较大时，算法的效率相当低。因此算法的平均时间复杂度为 $O(n)$ 。

139



139

## 3 顺序线性表的删除

### 3 顺序线性表的删除

在线性表  $L = (a_1, \dots, a_{i-1}, a_i, a_{i+1}, \dots, a_n)$  中删除结点 $a_i (1 \leq i \leq n)$ ，使其成为线性表：

$$L = (a_1, \dots, a_{i-1}, a_{i+1}, \dots, a_n)$$

#### 实现步骤

- (1) 将线性表 $L$ 中的第 $i+1$ 个至第 $n$ 个结点依次向前移动一个位置。
- (2) 线性表长度减1。

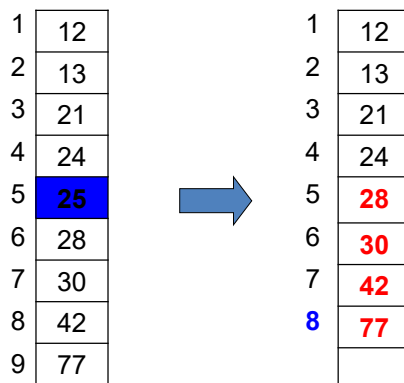
140



140

## 删除

删除顺序表中某个指定的元素的示意图如下：



## 算法描述

```
ElemType Delete_SqList(SqList *L, int i)
{
    int k; ElemType x;
    if (L->length==0)
        { printf("线性表L为空!\n"); return ERROR; }
    else if ( i<1||i>L->length )
        { printf("要删除的数据元素不存在!\n");
          return ERROR; }
    else { x=L->Elem_array[i-1]; /*保存结点的值*/
          for ( k=i; k<L->length; k++)
              L->Elem_array[k-1]=L->Elem_array[k];
          /* i位置以后的所有结点前移 */
          L->length--; return (x);
        }
}
```

在顺序表上做删除运算，平均要移动表上一半结点。当表长n较大时，算法的效率相当低。因此算法的平均时间复杂度为O(n)。

## 4 顺序线性表的查找定位删除

在线性表  $L = (a_1, a_2, \dots, a_n)$  中删除值为  $x$  的第一个结点。

### 实现步骤

- (1) 在线性表  $L$  查找值为  $x$  的第一个数据元素。
- (2) 将从找到的位置至最后一个结点依次向前移动一个位置。
- (3) 线性表长度减1。

## 算法描述

```
/* 删除线性表L中值为x的第一个结点 */
Status Locate_Delete_SqList(SqList *L, ElemType x)
{
    int i=0, k;
    while (i<L->length) /*查找值为x的第一个结点*/
    {
        if (L->Elem_array[i]!=x) i++;
        else
        {
            for (k=i+1; k<L->length; k++)
                L->Elem_array[k-1]=L->Elem_array[k];
            L->length--; break;
        }
    }
    if (i>L->length)
    {
        printf("要删除的数据元素不存在!\n");
        return ERROR;
    }
    return OK;
}
```



## 时间复杂度分析

时间主要耗费在数据元素的比较和移动操作上。

**首先**，在线性表L中查找值为x的结点是否存在；

**其次**，若值为x的结点存在，且在线性表L中的位置为i，则在线性表L中删除第i个元素。

设在线性表L删除数据元素概率为 $P_i$ ，不失一般性，设各个位置是等概率，则

$$P_i = 1/n。$$

◆ 比较的平均次数： $E_{compare} = \sum p_i * i \quad (1 \leq i \leq n)$

$$\therefore E_{compare} = (n + 1)/2。$$

◆ 删除时平均移动次数： $E_{delete} = \sum p_i * (n - i) \quad (1 \leq i \leq n)$

$$\therefore E_{delete} = (n - 1)/2。 \quad \text{平均时间复杂度：} E_{compare} + E_{delete} = n， \text{即为} O(n)$$



## 小结

线性表**顺序存储结构特点**：逻辑关系上相邻的两个元素在物理存储位置上也相邻；

**优点**：可以随机存取表中任一元素，方便快捷；

**缺点**：在插入或删除某一元素时，需要移动大量元素。

**解决问题的思路**：改用另一种线性存储方式：

**链式存储结构**



## 线性表的链式存储

**链式存储**：用一组任意的存储单元存储线性表中的数据元素。用这种方法存储的线性表简称**线性链表**。

存储链表中结点的一组任意的存储单元可以是连续的，也可以是不连续的，甚至是零散分布在内存中的任意位置上的。

链表中结点的逻辑顺序和物理顺序不一定相同。

为了正确表示结点间的逻辑关系，在存储每个结点值的同时，还必须存储指示其直接后继结点的地址(或位置)，称为指针(pointer)或链(link)，这两部分组成了链表中的结点结构，如下图所示。

链表是通过每个结点的指针域将线性表的n个结点按其逻辑次序链接在一起的。

每一个结只包含一个指针域的链表，称为单链表。

为操作方便，总是在链表的第一个结点之前附设一个头结点(头指针)head指向第一个结点。头结点的数据域可以不存储任何信息(或链表长度等信息)。

data	next
------	------

**data**：数据域，存放结点的值。**next**：指针域，存放结点的直接后继的地址。

图 链表结点结构

147



147

单链表是由表头唯一确定，因此单链表可以用头指针的名字来命名。

例1、线性表L=(bat, cat, eat, fat, hat)

其带头结点的单链表的逻辑状态和物理存储方式如图所示。

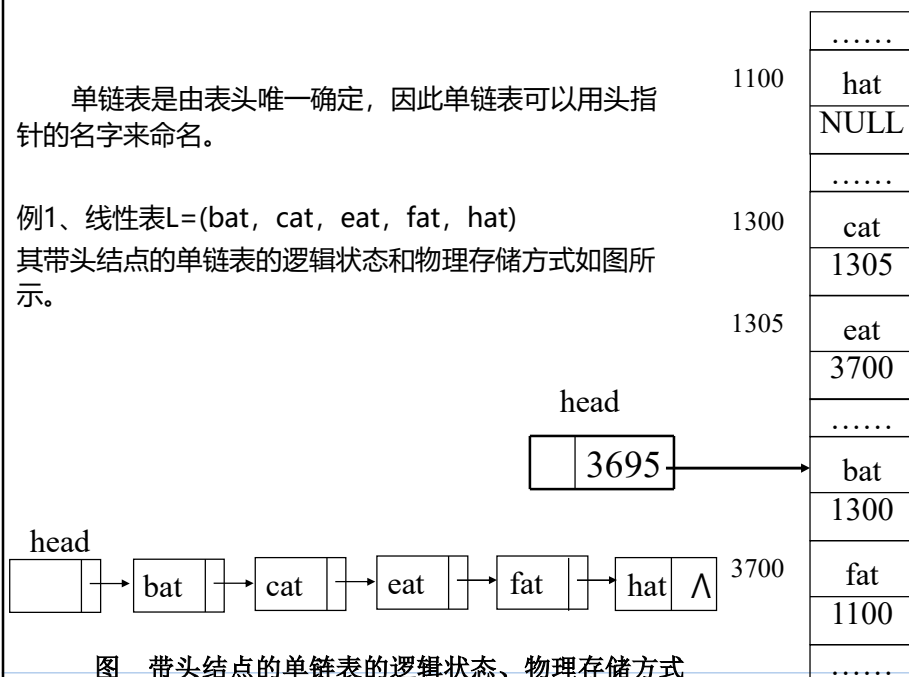


图 带头结点的单链表的逻辑状态、物理存储方式

148

148

## 单链表的实现

### 1 结点的描述与实现

C语言中用带指针的结构体类型来描述

```
typedef struct LNode
{ ElemType data; /*数据域，保存结点的值 */
  struct LNode *next; /*指针域*/
}LNode; /*结点的类型 */
```

### 2 结点的实现

结点是通过动态分配和释放来的实现，即需要时分配，不需要时释放。实现时是分别使用C语言提供的标准函数：malloc()，realloc()，sizeof()，free()。

**动态分配** p=(LNode\*)malloc(sizeof(LNode));

函数malloc分配了一个类型为LNode的结点变量的空间，并将其首地址放入指针变量p中。

**动态释放** free(p);

系统回收由指针变量p所指向的内存区。P必须是最近一次调用malloc函数时的返回值。

149



联航精英训练营

149

## 单线性链式的基本操作

### 1 建立单链表

假设线性表中结点的数据类型是整型，以32767作为结束标志。动态地建立单链表的常用方法有如下两种：头插入法，尾插入法。

#### (1) 头插入法建表

从一个空表开始，重复读入数据，生成新结点，将读入数据存放到新结点的数据域中，然后将新结点插入到当前链表的表头上，直到读入结束标志为止。即每次插入的结点都作为链表的第一个结点。

150



联航精英训练营

150

### 算法描述

```
LNode *create_LinkList(void)
/* 头插入法创建单链表,链表的头结点head作为返回值 */
{
    int data;
    LNode *head, *p;
    head = (LNode *) malloc( sizeof(LNode));
    head->next = NULL; /* 创建链表的表头结点head */
    while (1)
    {
        scanf( "%d" , &data);
        if (data == 32767) break;
        p = (LNode *) malloc(sizeof(LNode));
        p->data = data; /* 数据域赋值 */
        p->next = head->next;
        head->next = p; /* 钩链, 新创建的结点总是作为第一个结点 */
    }
    return (head);
}
```

151

151

## (2) 尾插入法建表

头插入法建立链表虽然算法简单,但生成的链表中结点的次序和输入的顺序相反。若希望二者次序一致,可采用尾插法建表。该方法是将新结点插入到当前链表的表尾,使其成为当前链表的尾结点。

无论是哪种插入方法,如果要插入建立的单线性链表的结点是 $n$ 个,算法的时间复杂度均为 $O(n)$ 。

对于单链表,无论是哪种操作,只要涉及到钩链(或重新钩链),如果没有明确给出直接后继,钩链(或重新钩链)的次序必须是“先右后左”。

152

152

### 算法描述:

```
LNode *create_LinkList(void)
/* 尾插入法创建单链表,链表的头结点head作为返回值 */
{ int data ;
  LNode *head, *p, *q;
  head=p=(LNode *)malloc(sizeof(LNode));
  p->next=NULL; /* 创建单链表的表头结点head */
  while (1)
  { scanf( "%d" ,& data);
    if (data==32767) break ;
    q= (LNode *)malloc(sizeof(LNode));
    q->data=data; /* 数据域赋值 */

    /*钩链, 新创建的结点总是作为最后一个结点*/
    q->next=p->next; p->next=q; p=q ;
  }
  return (head);
}
```

153

153

## 2 单链表的查找

(1) **按序号查找** 取单链表中的第*i*个元素。

对于单链表,不能象顺序表中那样直接按序号*i*访问结点,而只能从链表的头结点出发,沿链域next逐个结点往下搜索,直到搜索到第*i*个结点为止。因此,链表不是随机存取结构。

设单链表的长度为*n*,要查找表中第*i*个结点,仅当 $1 \leq i \leq n$ 时,*i*的值是合法的。

### 算法描述

```
ElemType Get_Elem(LNode *L , int i)
{ int j; LNode *p;
  p=L->next; j=1; /* 使p指向第一个结点 */
  while (p!=NULL && j<i)
  { p=p->next; j++; } /* 移动指针p , j计数 */
  if (j!=i) return(-32768);
  else return(p->data);
  /* p为NULL 表示i太大; j>i表示i为0 */
}
```

移动指针*p*的频度:

$i < 1$ 时: 0次;  $i \in [1, n]$ :  $i-1$ 次;  $i > n$ :  $n$ 次。

∴时间复杂度:  $O(n)$ 。

154

154

## (2) 按值查找

按值查找是在链表中，查找是否有结点值等于给定值key的结点？若有，则返回首次找到的值为key的结点的存储位置；否则返回NULL。查找时从开始结点出发，沿链表逐个将结点的值和给定值key作比较。

### 算法描述

```
LNode *Locate_Node(LNode *L, int key)
/* 在以L为头结点的单链表中查找值为key的第一个结点 */
{
    LNode *p=L->next;
    while (p!=NULL&& p->data!=key) p=p->next;
    if (p->data==key) return p;
    else
    {
        printf("所要查找的结点不存在!!\n");
        return(NULL);
    }
}
```

算法的执行与形参key有关，平均时间复杂度为 $O(n)$ 。

155



155

## 3 单链表的插入

插入运算是将值为e的新结点插入到表的第i个结点的位置上，即插入到 $a_{i-1}$ 与 $a_i$ 之间。因此，必须首先找到 $a_{i-1}$ 所在的结点p，然后生成一个数据域为e的新结点q，q结点作为p的直接后继结点。

**算法描述：**/\* 在以L为头结点的单链表的第i个位置插入值为e的结点 \*/

```
void Insert_LNode(LNode *L, int i, ElemType e)
{
    int j=0; LNode *p, *q;
    p=L->next;
    while (p!=NULL&& j<i-1)
    {
        p=p->next; j++;
    }
    if (j!=i-1) printf("i太大或i为0!!\n");
    else
    {
        q=(LNode *)malloc(sizeof(LNode));
        q->data=e; q->next=p->next;
        p->next=q;
    }
}
```

设链表的长度为n，合法的插入位置是 $1 \leq i \leq n$ 。算法的时间主要耗费移动指针p上，故时间复杂度亦为 $O(n)$ 。

156



156

## 4 单链表的删除

### (1) 按序号删除

删除单链表中的第 $i$ 个结点。

为了删除第 $i$ 个结点 $a_i$ ，必须找到结点的存储地址。该存储地址是在其直接前趋结点 $a_{i-1}$ 的next域中，因此，必须首先找到 $a_{i-1}$ 的存储位置 $p$ ，然后令 $p \rightarrow \text{next}$ 指向 $a_i$ 的直接后继结点，即把 $a_i$ 从链上摘下。最后释放结点 $a_i$ 的空间，将其归还给“存储池”。

设单链表长度为 $n$ ，则删去第 $i$ 个结点仅当 $1 \leq i \leq n$ 时是合法的。则当 $i = n + 1$ 时，虽然被删结点不存在，但其前趋结点却存在，是终端结点。故判断条件之一是 $p \rightarrow \text{next} \neq \text{NULL}$ 。显然此算法的时间复杂度也是 $O(n)$ 。

#### 算法描述

```
void Delete_LinkList(LNode *L, int i)
/* 删除以L为头结点的单链表中的第i个结点 */
{ int j=1; LNode *p, *q;
  p=L; q=L->next;
  while ( p->next!=NULL&& j<i)
    { p=q; q=q->next; j++; }
  if (j!=i) printf( "i太大或为0!!\n" );
  else
    { p->next=q->next; free(q); }
}
```

157



联航精英训练营

157

## 4 单链表的删除

### (2) 按值删除

删除单链表中值为key的第一个结点。

与按值查找相类似，首先要查找值为key的结点是否存在？若存在，则删除；否则返回NULL。

#### 算法描述

```
void Delete_LinkList(LNode *L, int key)
/* 删除以L为头结点的单链表中值为key的第一个结点 */
{ LNode *p=L, *q=L->next;
  while ( q!=NULL&& q->data!=key)
    { p=q; q=q->next; }
  if (q->data==key)
    { p->next=q->next; free(q); }
  else
    printf( "所要删除的结点不存在!!\n" );
}
```

158



联航精英训练营

158

算法的执行与形参key有关，平均时间复杂度为 $O(n)$ 。

从上面的讨论可以看出，链表上实现插入和删除运算，无需移动结点，仅需修改指针。解决了顺序表的插入或删除操作需要移动大量元素的问题。

#### 变形之一：

删除单链表中值为key的所有结点。

与按值查找相类似，但比前面的算法更简单。

**基本思想：**从单链表的第一个结点开始，对每个结点进行检查，若结点的值为key，则删除之，然后检查下一个结点，直到所有的结点都检查。

#### 算法描述

```
void Delete_LinkList_Node(LNode *L, int key)
/* 删除以L为头结点的单链表中值为key的第一个结点 */
{   LNode *p=L, *q=L->next;
    while ( q!=NULL)
        { if (q->data==key)
            { p->next=q->next; free(q); q=p->next; }
          else
            { p=q; q=q->next; }
        }
}
```

159

159

#### 变形之二：

删除单链表中所有值重复的结点，使得所有结点的值都不相同。

与按值查找相类似，但比前面的算法更复杂。

**基本思想：**从单链表的第一个结点开始，对每个结点进行检查：检查链表中该结点的所有后继结点，只要有值和该结点的值相同，则删除之；然后检查下一个结点，直到所有的结点都检查。

#### 算法描述

```
void Delete_Node_value(LNode *L)
/* 删除以L为头结点的单链表中所有值相同的结点 */
{   LNode *p=L->next, *q, *ptr;
    while ( p!=NULL) /* 检查链表中所有结点 */
        { *q=p, *ptr=p->next;
          /* 检查结点p的所有后继结点ptr */
          while (ptr!=NULL)
              { if (ptr->data==p->data)
                  { q->next=ptr->next; free(ptr);
                    ptr=q->next; }
                else { q=ptr; ptr=ptr->next; }
              }
          p=p->next;
        }
}
```

160

160



## 5 单链表的合并

设有两个有序的单链表，它们的头指针分别是La、Lb，将它们合并为以Lc为头指针的有序链表。合并前的示意图如图所示。

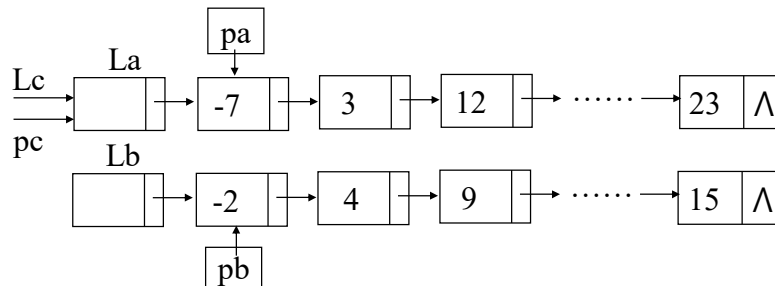


图 两个有序的单链表La，Lb的初始状态

161

161

合并了值为-7，-2的结点后示意图如图所示。

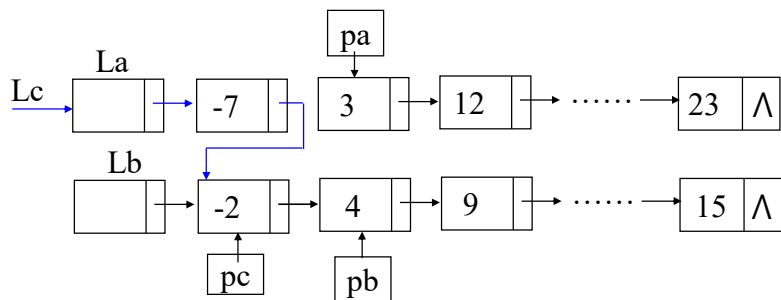


图 合并了值为-7，-2的结点后的状态

### 算法说明

算法中pa，pb分别是待考察的两个链表的当前结点，pc是合并过程中合并的链表的最后一个结点。

### 算法分析

若La，Lb两个链表的长度分别是m，n，则链表合并的时间复杂度为 $O(m+n)$ 。

162

162

### 算法描述

```
/* 合并以La, Lb为头结点的两个有序单链表 */
LNode *Merge_LinkList(LNode *La, LNode *Lb)
{
    LNode *Lc, *pa, *pb, *pc, *ptr;
    Lc=La; pc=La; pa=La->next; pb=Lb->next;
    while (pa!=NULL && pb!=NULL)
    {
        /* 将pa所指的结点合并, pa指向下一个结点 */
        if (pa->data<pb->data)
        {
            pc->next=pa; pc=pa; pa=pa->next;
        }
        /* 将pb所指的结点合并, pb指向下一个结点 */
        if (pa->data>pb->data)
        {
            pc->next=pb; pc=pb; pb=pb->next;
        }
        /* 将pa所指的结点合并, pb所指结点删除 */
        if (pa->data==pb->data)
        {
            pc->next=pa; pc=pa; pa=pa->next;
            ptr=pb; pb=pb->next; free(ptr);
        }
    }
    if (pa!=NULL) pc->next=pa;
    else pc->next=pb; /*将剩余的结点链上*/
    free(Lb);
    return(Lc);
}
```

163

163

## 循环链表

**循环链表(Circular Linked List)**: 是一种头尾相接的链表。其特点是最后一个结点的指针域指向链表的头结点, 整个链表的指针域链接成一个环。

从循环链表的任意一个结点出发都可以找到链表中的其它结点, 使得表处理更加方便灵活。

下图是带头结点的单循环链表的示意图。

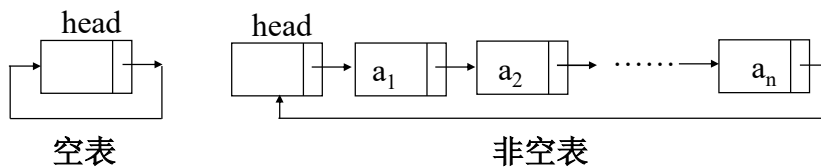


图 单循环链表示意图

164

164

## 循环链表的操作

### 循环链表的操作

对于单循环链表，除链表的合并外，其它的操作和单线性链表基本上一致，仅仅需要在单线性链表操作算法基础上作以下简单修改：

- (1) 判断是否是空链表：head->next==head；
- (2) 判断是否是表尾结点：p->next==head；

## 双向链表

**双向链表(Double Linked List)** :指的是构成链表的每个结点中设立两个指针域：一个指向其直接前趋的指针域prior，一个指向其直接后继的指针域next。这样形成的链表中有两个方向不同的链，故称为**双向链表**。

和单链表类似，双向链表一般增加头指针也能使双链表上的某些运算变得方便。

将头结点和尾结点链接起来也能构成循环链表，并称之为双向循环链表。

双向链表是为了克服单链表的单向性的缺陷而引入的。

## 双向链表

### 1 双向链表的结点及其类型定义

双向链表的结点的类型定义如下。其结点形式如图所示，带头结点的双向链表的形式如图所示。

```
typedef struct DulNode
{
    ElemType data;
    struct DulNode *prior, *next;
}DulNode;
```

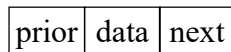
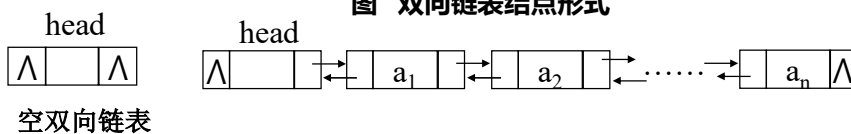


图 双向链表结点形式



空双向链表

图 带头结点的双向链表形式

非空双向链表

167



167

双向链表结构具有对称性，设p指向双向链表中的某一结点，则其对称性可用下式描述：

$$(p \rightarrow \text{prior}) \rightarrow \text{next} = p = (p \rightarrow \text{next}) \rightarrow \text{prior};$$

结点p的存储位置存放在其直接前趋结点p->prior的直接后继指针域中，同时也存放在其直接后继结点p->next的直接前趋指针域中。

### 2 双向链表的基本操作

(1) **双向链表的插入** 将值为e的结点插入双向链表中。插入前后链表的变化如图所示。

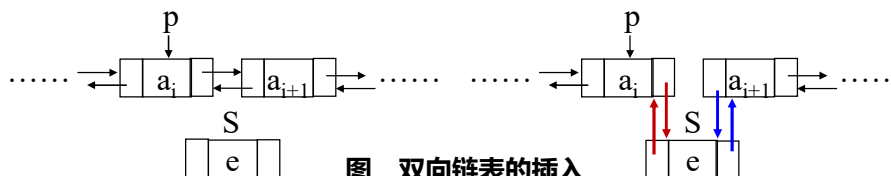


图 双向链表的插入

168

168

① 插入时仅仅指出直接前驱结点，钩链时必须注意先后次序是：“先右后左”。  
部分语句组如下：

```
S=(DulNode *)malloc(sizeof(DulNode));  
S->data=e;  
S->next=p->next; p->next->prior=S;  
p->next=S; S->prior=p; /* 钩链次序非常重要 */
```

② 插入时同时指出直接前驱结点p和直接后继结点q，钩链时无须注意先后次序。  
部分语句组如下：

```
S=(DulNode *)malloc(sizeof(DulNode));  
S->data=e;  
p->next=S; S->next=q;  
S->prior=p; q->prior=S;
```

169

169

## (2) 双向链表的结点删除

设要删除的结点为p，删除时可以不引入新的辅助指针变量，可以直接先断链，再释放结点。部分语句组如下：

```
p->prior->next=p->next;  
p->next->prior=p->prior;  
free(p);
```

注意：

与单链表的插入和删除操作不同的是，在双向链表中插入和删除必须同时修改两个方向上的指针域的指向。

170

170

## 一元多项式的表示和相加

### 1 一元多项式的表示

一元多项式  $p(x)=p_0+p_1x+p_2x^2+\dots+p_nx^n$ ，由  $n+1$  个系数唯一确定。则在计算机中可用线性表  $(p_0, p_1, p_2, \dots, p_n)$  表示。既然是线性表，就可以用顺序表和链表来实现。两种不同实现方式的元素类型定义如下：

#### (1) 顺序存储表示的类型

```
typedef struct
{ float coef; /*系数部分*/
  int expn; /*指数部分*/
} ElemType;
```

#### (2) 链式存储表示的类型

```
typedef struct ploy
{ float coef; /*系数部分*/
  int expn; /*指数部分*/
  struct ploy *next;
} Ploy;
```

## 一元多项式的相加

不失一般性，设有两个一元多项式：

$$P(x)=p_0+p_1x+p_2x^2+\dots+p_nx^n,$$

$$Q(x)=q_0+q_1x+q_2x^2+\dots+q_mx^m \quad (m<n)$$

$$R(x)=P(x)+Q(x)$$

$R(x)$  由线性表  $R((p_0+q_0), (p_1+q_1), (p_2+q_2), \dots, (p_m+q_m), \dots, p_n)$  唯一表示。

## 一元多项式相加

### (1) 顺序存储表示的相加

线性表的定义

```
typedef struct  
{ ElemType a[MAX_SIZE];  
  int length;  
}Sqlist;
```

用顺序表示的相加非常简单。访问第5项可直接访问: L.a[4].coef, L.a[4].expn

### (2) 链式存储表示的相加

当采用链式存储表示时, 根据结点类型定义, 凡是系数为0的项不在链表中出现, 从而可以大大减少链表的长度。

173



173

## 一元多项式相加的实质

**一元多项式相加的实质是:**

**指数不同:** 是链表的合并。

**指数相同:** 系数相加, 和为0, 去掉结点, 和不为0, 修改结点的系数域。

### 算法之一:

就在原来两个多项式链表的基础上进行相加, 相加后原来两个多项式链表就不存在了。当然再要对原来两个多项式进行其它操作就不允许了。

### 算法之二:

对两个多项式链表进行相加, 生成一个新的相加后的结果多项式链表, 原来两个多项式链表依然存在, 不发生改变, 如果要再对原来两个多项式进行其它操作也不影响。

174



174

## 线性表小结

■ 数组

■ 链表

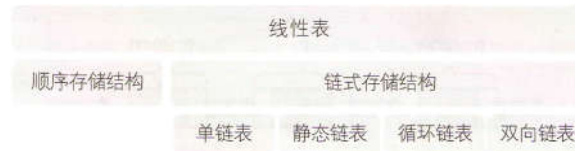
头指针链表

头节点链表

单向链表

循环链表

双向链表



扩展：

一元多项式的计算

175

175

专业铸就品质 梦想成就未来

The Specialty Casts Quality, the Dream Gains Future.

176

176



## 第五章 栈与队列

177



177

### 栈与队列

栈和队列是两种应用非常广泛的数据结构，它们都来自线性表数据结构，都是“**操作受限**”的线性表。

栈在计算机的实现有多种方式：

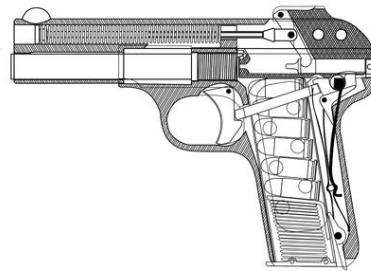
- ◆ **硬堆栈**：利用CPU中的某些寄存器组或类似的硬件或使用内存的特殊区域来实现。这类堆栈容量有限，但速度很快；
- ◆ **软堆栈**：这类堆栈主要在内存中实现。堆栈容量可以达到很大。在实现方式上，又有**动态方式**和**静态方式**两种。

178



178

## 栈



179

联航精英训练营

179

## 堆栈

- 堆栈(stack)是一组元素的集合，类似于数组，不同之处在于，数组可以按**下标随机访问**，这次访问a[5]下次可以访问a[1]，但是堆栈的访问规则被限制为**Push**和**Pop**两种操作，Push（入栈或压栈）向栈顶添加元素，Pop（出栈或弹出）则取出当前栈顶的元素，也就是说，**只能访问栈顶元素而不能访问栈中其它元素**。如果所有元素的类型相同，堆栈的存储也可以用数组来实现，访问操作可以通过函数接口提供。  
(LIFO)

180

联航精英训练营

180

## 栈的概念

**栈(Stack)**: 是限制在表的一端进行插入和删除操作的线性表。又称为后进先出 LIFO (Last In First Out)或先进后出FILO (First In Last Out)线性表。

**栈顶(Top)**: 允许进行插入、删除操作的一端, 又称为表尾。用栈顶指针(top)来指示栈顶元素。

**栈底(Bottom)**: 是固定端, 又称为表头。

**空栈**: 当表中没有元素时称为空栈。

设栈 $S=(a_1, a_2, \dots, a_n)$ , 则 $a_1$ 称为栈底元素,  $a_n$ 为栈顶元素, 如图所示。

栈中元素按 $a_1, a_2, \dots, a_n$ 的次序进栈, 退栈的第一个元素应为栈顶元素。即栈的修改是按后进先出的原则进行的。

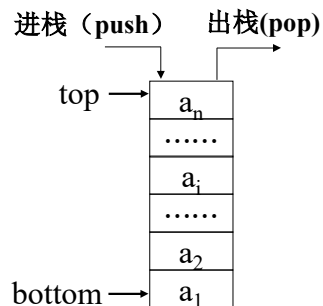


图 顺序栈示意图

## 栈的抽象数据类型定义

ADT Stack{

数据对象:  $D = \{ a_i | a_i \in \text{ElemSet}, i=1,2,\dots,n, n \geq 0 \}$

数据关系:  $R = \{ \langle a_{i-1}, a_i \rangle | a_{i-1}, a_i \in D, i=2,3,\dots,n \}$

基本操作: 初始化、进栈、出栈、取栈顶元素等

} ADT Stack

## 栈的顺序存储表示

栈的顺序存储结构简称为顺序栈，和线性表相类似，用一维数组来存储栈。根据数组是否可以根据需要增大，又可分为静态顺序栈和动态顺序栈。

- ◆ 静态顺序栈实现简单，但不能根据需要增大栈的存储空间；
- ◆ 动态顺序栈可以根据需要增大栈的存储空间，但实现稍为复杂。

## 栈的动态顺序存储表示

采用动态一维数组来存储栈。所谓动态，指的是栈的大小可以根据需要增加。

- ◆ 用bottom表示栈底指针，栈底固定不变的；栈顶则随着进栈和退栈操作而变化。用top(称为栈顶指针)指示当前栈顶位置。
- ◆ 用top=bottom作为栈空的标记，每次top指向栈顶数组中的下一个存储位置。
- ◆ 结点进栈：首先将数据元素保存到栈顶(top所指的当前位置)，然后执行top加1，使top指向栈顶的下一个存储位置；

◆ **结点出栈**：首先执行top减1，使top指向栈顶元素的存储位置，然后将栈顶元素取出。

下图是一个动态栈的变化示意图。

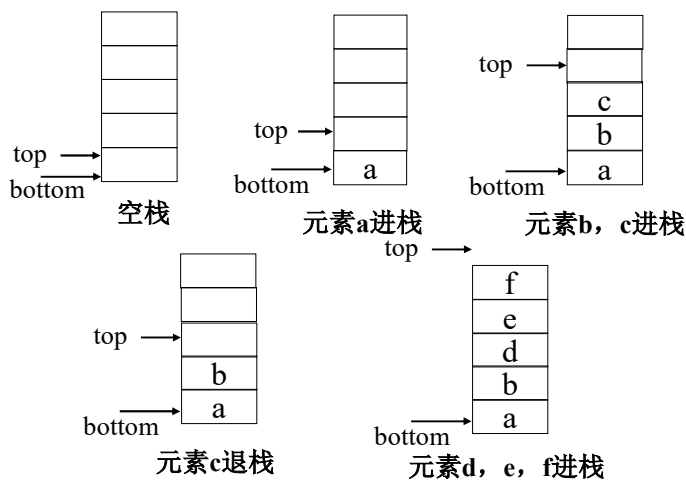


图 (动态)堆栈变化示意图

185

185

## 基本操作的实现

### 1. 栈的类型定义

```
#define STACK_SIZE 100 /* 栈初始向量大小 */
#define STACKINCREMENT 10 /* 存储空间分配增量 */
typedef int ElemType;
typedef struct sqstack
{
    ElemType *bottom; /* 栈不存在时值为NULL */
    ElemType *top; /* 栈顶指针 */
    int stacksize; /* 当前已分配空间，以元素为单位 */
}SqStack;
```

### 2. 栈的初始化

```
Status Init_Stack(void)
{
    SqStack S;
    S.bottom=(ElemType *)malloc(STACK_SIZE *sizeof(ElemType));
    if (! S.bottom) return ERROR;
    S.top=S.bottom; /* 栈空时栈顶和栈底指针相同 */
    S.stacksize=STACK_SIZE;
    return OK;
}
```

186

186

### 3. 压栈 (元素进栈)

```
Status push(SqStack S, ElemType e)
{ if (S.top-S.bottom>=S.stacksize-1)
    { S.bottom=(ElemType *)realloc((S.STACKINCREMENT+STACK_SIZE)
    *sizeof(ElemType)); /* 栈满, 追加存储空间 */
    if (! S.bottom) return ERROR;
    S.top=S.bottom+S.stacksize;
    S.stacksize+=STACKINCREMENT;
    }
    *S.top=e; S.top++; /* 栈顶指针加1, e成为新的栈顶 */
    return OK;
}
```

### 4. 弹栈 (元素出栈)

```
Status pop( SqStack S, ElemType *e )
/*弹出栈顶元素*/
{ if ( S.top== S.bottom )
    return ERROR; /* 栈空, 返回失败标志 */
    S.top--; e=*S.top;
    return OK;
}
```

187

187

## 栈的静态顺序存储表示

采用静态一维数组来存储栈。

栈底固定不变的, 而栈顶则随着进栈和退栈操作变化的,

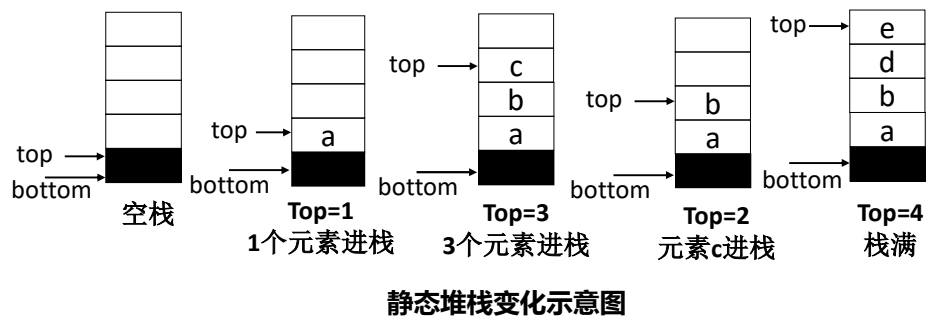
- ◆ 栈底固定不变的; 栈顶则随着进栈和退栈操作而变化, 用一个整型变量top (称为栈顶指针) 来指示当前栈顶位置。
- ◆ 用top=0表示栈空的初始状态, 每次top指向栈顶在数组中的存储位置。
- ◆ **结点进栈**: 首先执行top加1, 使top指向新的栈顶位置, 然后将数据元素保存到栈顶(top所指的当前位置)。
- ◆ **结点出栈**: 首先把top指向的栈顶元素取出, 然后执行top减1, 使top指向新的栈顶位置。

188

188

## 静态栈的动态变化

若栈的数组有Maxsize个元素，则 $\text{top} = \text{Maxsize} - 1$ 时栈满。下图是一个大小为5的栈的变化示意图。



189

189

## 基本操作的实现

### 1 栈的类型定义

```
#define MAX_STACK_SIZE 100 /* 栈向量大小 */
typedef int ElemType;
typedef struct sqstack
{
    ElemType stack_array[MAX_STACK_SIZE];
    int top;
}SqStack;
```

### 2 栈的初始化

```
SqStack Init_Stack(void)
{
    SqStack S;
    S.bottom=S.top=0; return(S);
}
```

190

190

## 基本操作的实现

### 3 压栈(元素进栈)

```
Status push(SqStack S, ElemType e)
/* 使数据元素e进栈成为新的栈顶 */
{ if (S.top==MAX_STACK_SIZE-1)
    return ERROR; /* 栈满, 返回错误标志 */
    S.top++; /* 栈顶指针加1 */
    S.stack_array[S.top]=e; /* e成为新的栈顶 */
    return OK; /* 压栈成功 */
}
```

### 4 弹栈(元素出栈)

```
Status pop(SqStack S, ElemType *e)
/*弹出栈顶元素*/
{ if (S.top==0)
    return ERROR; /* 栈空, 返回错误标志 */
    *e=S.stack_array[S.top];
    S.top--;
    return OK;
}
```

## 栈的上溢与下溢

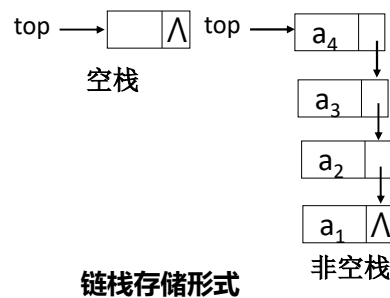
当栈满时做进栈运算必定产生空间溢出, 简称“**上溢**”。上溢是一种出错状态, 应设法避免。

当栈空时做退栈运算也将产生溢出, 简称“**下溢**”。下溢则可能是正常现象, 因为栈在使用时, 其初态或终态都是空栈, 所以下溢常用来作为控制转移的条件。



## 栈的链式表示

- 栈的**链式存储结构**称为**链栈**，是运算受限的单链表。其插入和删除操作只能在表头位置上进行。因此，链栈没有必要像单链表那样附加头结点，栈顶指针top就是链表的头指针。



193

193

## 2 链栈基本操作的实现

链栈的结点类型说明如下：

```
typedef struct Stack_Node
{ ElemType data;
  struct Stack_Node *next;
} Stack_Node;
```

### (1) 栈的初始化

```
Stack_Node *Init_Link_Stack(void)
{ Stack_Node *top;
  top=(Stack_Node *)malloc(sizeof(Stack_Node));
  top->next=NULL;
  return(top);
}
```

194

194

## (2) 压栈(元素进栈)

Status push(Stack\_Node \*top, ElemType e)

```
{ Stack_Node *p;  
  p=(Stack_Node *)malloc(sizeof(Stack_Node));  
  if (!p) return ERROR; /* 申请新结点失败, 返回错误标志 */  
  p->data=e;  
  p->next=top->next;  
  top->next=p; /* 钩链 */  
  return OK;  
}
```

## (3) 弹栈(元素出栈)

Status pop(Stack\_Node \*top, ElemType \*e)

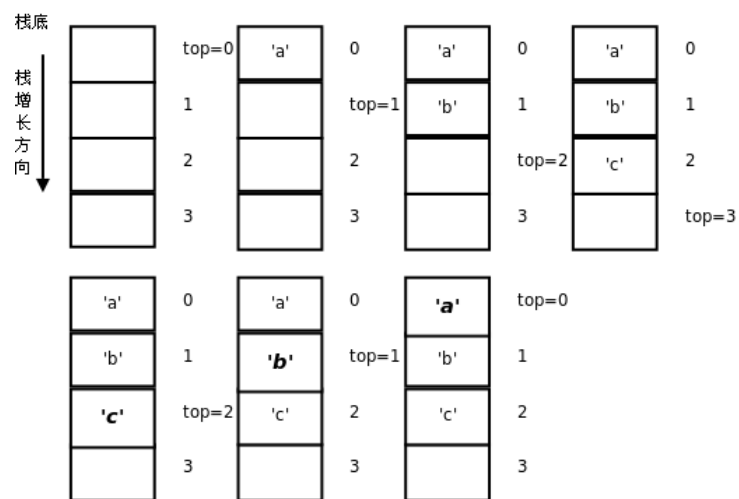
```
{ Stack_Node *p;  
  ElemType e;  
  if (top->next==NULL)  
    return ERROR; /* 栈空, 返回错误标志 */  
  p=top->next; e=p->data; /* 取栈顶元素 */  
  top->next=p->next; /* 修改栈顶指针 */  
  free(p);  
  return OK;  
}
```

195

195

## 用堆栈实现倒序打印

### ■ 过程分析



196

196

## 数制转换

十进制整数N向其它进制数d(二、八、十六)的转换是计算机实现计算的基本问题。

**转换法则：**该转换法则对应于一个简单算法原理：

$$n = (n \text{ div } d) * d + n \text{ mod } d$$

其中：div为整除运算,mod为求余运算

例如  $(1348)_{10} = (2504)_8$ ，其运算过程如下：

n	n div 8	n mod 8
1348	168	4
168	21	0
21	2	5
2	0	2

197



197

## 采用静态顺序栈方式实现

```
void conversion(int n, int d)
/*将十进制整数N转换为d(2或8)进制数*/
{ SqStack S; int k, *e;
  S=Init_Stack();
  while (n>0) { k=n%d; push(S, k); n=n/d; }
  /* 求出所有的余数, 进栈 */
  while (S.top!=0) /* 栈不空时出栈, 输出 */
  { pop(S, e);
    printf( "%1d" , *e);
  }
}
```

198



198

## 括号匹配问题

在文字处理软件或编译程序设计时，常常需要检查一个字符串或一个表达式中的括号是否相匹配？

**匹配思想：**从左至右扫描一个字符串(或表达式)，则**每个右括号将与最近遇到的那个左括号相匹配**。则可以在从左至右扫描过程中把所遇到的左括号存放到堆栈中。每当遇到一个右括号时，就将它与栈顶的左括号(如果存在)相匹配，同时从栈顶删除该左括号。

**算法思想：**设置一个栈，当读到左括号时，左括号进栈。当读到右括号时，则从栈中弹出一个元素，与读到的左括号进行匹配，若匹配成功，继续读入；否则匹配失败，返回FALSE。



```
int Match_Brackets( )
{ char ch, x;
  scanf( "%c" , &ch);
  while (asc(ch)!=13)
  { if ((ch== '(' )||(ch== '[' )) push(S, ch);
    else if (ch== ']' )
    { x=pop(S);
      if (x!= '[' )
      { printf( " " '[' 括号不匹配" );
        return FLASE ; } }
    else if (ch== ')' )
    { x=pop(S);
      if (x!= '(' )
      { printf( " " '(' 括号不匹配" );
        return FLASE ;}
    }
  }
  if (S.top!=0)
  { printf( "括号数量不匹配! " );
    return FLASE ;
  }
  else return TRUE ;
}
```

## 栈与递归调用的实现

栈的另一个重要应用是在程序设计语言中实现递归调用。

**递归调用**：一个函数(或过程)直接或间接地调用自己本身，简称**递归**(Recursive)。

**递归**是程序设计中的一个强有力的工具。因为递归函数结构清晰，程序易读，正确性很容易得到证明。

为了使递归调用不至于无终止地进行下去，实际上有效的递归调用函数(或过程)应包括两部分：**递推规则(方法)**，**终止条件**。

例如：求n!

$$\text{Fact}(n)=\begin{cases} 1 & \text{当}n=0\text{时} \quad \text{终止条件} \\ n*\text{fact}(n-1) & \text{当}n>0\text{时} \quad \text{递推规则} \end{cases}$$

为保证递归调用正确执行，系统设立一个“**递归工作栈**”，作为整个递归调用过程期间使用的数据存储空间。

每一层递归包含的信息如：**参数、局部变量、上一层的返回地址**构成一个“**工作记录**”。每进入一层递归，就产生一个新的工作记录压入栈顶；每退出一层递归，就从栈顶弹出一个工作记录。

201



201

## 从被调函数返回调用函数的一般步骤

从被调函数返回调用函数的一般步骤：

1. 若栈为空，则执行正常返回。
2. 从栈顶弹出一个工作记录。
3. 将“工作记录”中的参数值、局部变量值赋给相应的变量；读取返回地址。
4. 将函数值赋给相应的变量。
5. 转移到返回地址。

202



202

## 迷宫问题

```
int maze[5][5] = {  
    0, 1, 0, 0, 0,  
    0, 1, 0, 1, 0,  
    0, 0, 0, 1, 0,  
    0, 1, 1, 1, 0,  
    0, 0, 0, 1, 0,  
};
```

它表示一个迷宫，其中的1表示墙壁，0表示可以走的路，只能横着走或竖着走，不能斜着走，要求程序找出从左上角到右下角的路线。

## 堆栈与深度优先搜索

### ■ 用堆栈走迷宫:

将起点标记为已走过并压栈;

```
while (栈非空) {
```

```
    从栈顶弹出一个点p;
```

```
    if (p这个点是终点)
```

```
        break;
```

```
    否则沿右、下、左、上四个方向探索相邻的点
```

```
    if (和p相邻的点有路可走，并且还没走过)
```

```
        将相邻的点标记为已走过并压栈，它的前趋就是p点;
```

```
}
```

```
if (p点是终点) {
```

```
    打印p点的座标;
```

```
    while (p点有前趋) {
```

```
        p点 = p点的前趋;
```

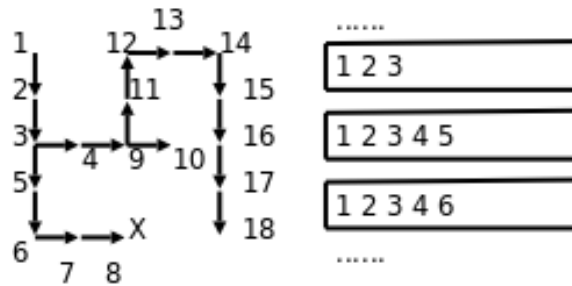
```
        打印p点的座标;
```

```
    }
```

```
} else
```

```
    没有路线可以到达终点;
```

## 深度优先搜索



这种搜索算法的特点是：每次探索完各个方向相邻的点之后，取其中一个相邻的点走下去，一直走到无路可走了再退回来，取另一个相邻的点再走下去。这称为**深度优先搜索**（DFS, Depth First Search）

205

联航精英训练营

205

## 队列



206

## 队列

### 1 队列的基本概念

**队列(Queue)**：也是运算受限的线性表。是一种**先进先出**(First In First Out，简称**FIFO**)的线性表。只允许在表的一端进行插入，而在另一端进行删除。

**队首(front)**：允许进行删除的一端称为队首。

**队尾(rear)**：允许进行插入的一端称为队尾。

例如：排队购物。操作系统中的作业排队。先进入队列的成员总是先离开队列。

队列中没有元素时称为空队列。在空队列中依次加入元素 $a_1, a_2, \dots, a_n$ 之后， $a_1$ 是队首元素， $a_n$ 是队尾元素。显然退出队列的次序也只能是 $a_1, a_2, \dots, a_n$ ，即队列的修改是依**先进先出**的原则进行的，如图所示。

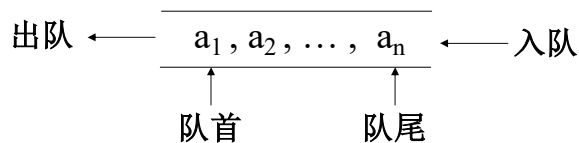


图 队列示意图

207

207

## 2 队列的抽象数据类型定义

ADT Queue{

**数据对象**： $D = \{ a_i | a_i \in \text{ElemSet}, i=1, 2, \dots, n, n \geq 0 \}$

**数据关系**： $R = \{ \langle a_{i-1}, a_i \rangle | a_{i-1}, a_i \in D, i=2, 3, \dots, n \}$

约定 $a_1$ 端为队首， $a_n$ 端为队尾。

**基本操作**：

Create()：创建一个空队列；

EmptyQue()：若队列为空，则返回true，否则返回false；

.....

InsertQue(x)：向队尾插入元素x；

DeleteQue(x)：删除队首元素x；

} ADT Queue

208

208



## 队列的顺序表示和实现

利用一组连续的存储单元(一维数组)依次存放从队首到队尾的各个元素,称为顺序队列。

对于队列,和顺序栈相类似,也有动态和静态之分。本部分介绍的是静态顺序队列,其类型定义如下:

```
#define MAX_QUEUE_SIZE 100
typedef struct queue
{ ElemType Queue_array[MAX_QUEUE_SIZE];
  int front;
  int rear;
}SqQueue;
```

209



联航精英训练营

209

## 队列的顺序存储结构

- 设立一个队首指针front, 一个队尾指针rear, 分别指向队首和队尾元素。
  - ◆ 初始化: front=rear=0。
  - ◆ 入队: 将新元素插入rear所指的位置, 然后rear加1。
  - ◆ 出队: 删去front所指的元素, 然后加1并返回被删元素。
  - ◆ 队列为空: front==rear。
  - ◆ 队满: rear==MAX\_QUEUE\_SIZE-1或front==rear。

210



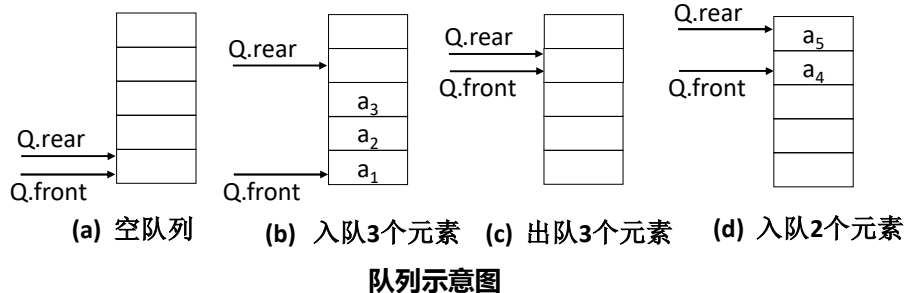
联航精英训练营

210

## 队列中的假溢出

在非空队列里，队首指针始终指向队头元素，而队尾指针始终指向队尾元素的下一位置。

顺序队列中存在“**假溢出**”现象。因为在入队和出队操作中，头、尾指针只增加不减小，致使被删除元素的空间永远无法重新利用。因此，尽管队列中实际元素个数可能远远小于数组大小，但可能由于尾指针已超出向量空间的上界而不能做入队操作。该现象称为**假溢出**。



211

211

## 循环队列

为充分利用向量空间，克服上述“假溢出”现象的方法是：将为队列分配的向量空间看成为一个首尾相接的圆环，并称这种队列为**循环队列**(Circular Queue)。

在循环队列中进行出队、入队操作时，队首、队尾指针仍要加1，朝前移动。只不过当队首、队尾指针指向向量上界(MAX\_QUEUE\_SIZE-1)时，其加1操作的结果是指向向量的下界0。

这种循环意义下的加1操作可以描述为：

```
if (i+1==MAX_QUEUE_SIZE) i=0;
```

```
else i++;
```

其中：i代表队首指针(front)或队尾指针(rear)

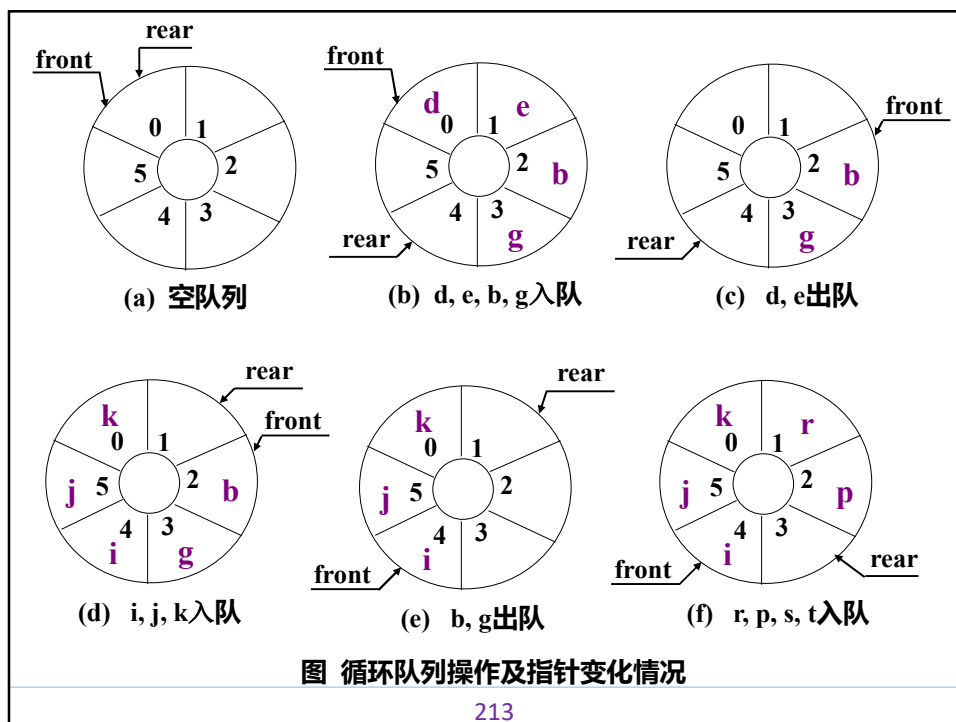
用模运算可简化为： $i=(i+1)\%MAX\_QUEUE\_SIZE$ ；

显然，为循环队列所分配的空间可以被充分利用，除非向量空间真的被队列元素全部占用，否则不会上溢。因此，真正实用的顺序队列是循环队列。

例：设有循环队列QU[0, 5]，其初始状态是front=rear=0，各种操作后队列的头、尾指针的状态变化情况如下图所示。

212

212



213

213

## 如何判断队列满还是空？

入队时尾指针向前追赶头指针，出队时头指针向前追赶尾指针，故队空和队满时头尾指针均相等。因此，无法通过 $front=rear$ 来判断队列“空”还是“满”。解决此问题的方法是：约定入队前，测试尾指针在循环意义下加1后是否等于头指针，若相等则认为队满。即：

- ◆ rear所指的单元始终为空。
- ◆ 循环队列为空： $front=rear$ 。
- ◆ 循环队列满： $(rear+1)\%MAX\_QUEUE\_SIZE = front$ 。

还有什么方法？

1. 设置一个变量表示当前队列存放多少有效元素
2. 设置一个变量标识上一次操作出队还是入队，当 $front=rear$ 时，标识为出队自然表示队空，入队则表示队满

214

214

## 循环队列的基本操作

### 1. 循环队列的初始化

```
SqQueue Init_CirQueue(void)
{ SqQueue Q;
  Q.front=Q.rear=0; return(Q);
}
```

### 2. 入队操作

```
/* 将数据元素e插入到循环队列Q的队尾 */
Status Insert_CirQueue(SqQueue Q, ElemType e)
{ if ((Q.rear+1)%MAX_QUEUE_SIZE== Q.front)
  return ERROR; /* 队满, 返回错误标志 */
  Q.Queue_array[Q.rear]=e; /* 元素e入队 */
  Q.rear=(Q.rear+1)% MAX_QUEUE_SIZE; /* 队尾指针向前移动 */
  return OK; /* 入队成功 */
}
```

215



联航精英训练营

215

### 3. 出队操作

```
/* 将循环队列Q的队首元素出队 */
Status Delete_CirQueue(SqQueue Q, ElemType *x)
{ if (Q.front+1== Q.rear)
  return ERROR; /* 队空, 返回错误标志 */
  *x=Q.Queue_array[Q.front]; /* 取队首元素 */
  Q.front=(Q.front+1)% MAX_QUEUE_SIZE; /* 队首指针向前移动 */
  return OK;
}
```

216

216

## 队列的链式表示和实现

### 1 队列的链式存储表示

队列的链式存储结构简称为**链队列**，它是限制仅在表头进行删除操作和表尾进行插入操作的单链表。

需要两类不同的结点：**数据元素结点**，队列的**队首指针**和**队尾指针**的结点，如图所示。

数据元素结点类型定义：

```
typedef struct Qnode
{ ElemType data;
  struct Qnode *next;
}QNode;
```

指针结点类型定义：

```
typedef struct link_queue
{ QNode *front, *rear;
}Link_Queue;
```

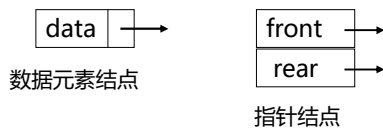


图 链队列结点示意图

217



联航精英训练营

217

## 2 链队运算及指针变化

链队的操作实际上是单链表的操作，只不过是删除在表头进行，插入在表尾进行。插入、删除时分别修改不同的指针。链队运算及指针变化如图所示。

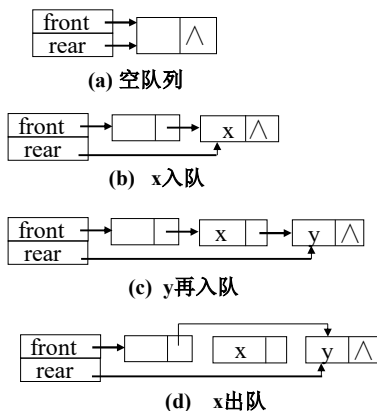


图 队列操作及指针变化

218



联航精英训练营

218

### 3 链队列的基本操作

#### (1) 链队列的初始化

```
LinkQueue *Init_LinkQueue(void)
{
    LinkQueue *Q; QNode *p;
    p=(QNode *)malloc(sizeof(QNode));          /* 开辟头结点 */
    p->next=NULL;
    Q=(LinkQueue *)malloc(sizeof(LinkQueue));    /* 开辟链队的指针结点 */
    Q.front=Q.rear=p;
    return(Q);
}
```

#### (2) 链队列的入队操作

在已知队列的队尾插入一个元素e，即修改队尾指针(Q.rear)。

```
Status Insert_CirQueue(LinkQueue *Q, ElemType e)
{
    p=(QNode *)malloc(sizeof(QNode));
    if (!p) return ERROR;          /* 申请新结点失败，返回错误标志 */
    p->data=e; p->next=NULL;        /* 形成新结点 */
    Q.rear->next=p; Q.rear=p;      /* 新结点插入到队尾 */
    return OK;
}
```

219



联航精英训练营

219

#### (3) 链队列的出队操作

```
Status Delete_LinkQueue(LinkQueue *Q, ElemType *x)
{
    QNode *p;
    if (Q.front==Q.rear) return ERROR; /* 队空 */
    p=Q.front->next; /* 取队首结点 */
    *x=p->data;
    Q.front->next=p->next; /* 修改队首指针 */
    if (p==Q.rear) Q.rear=Q.front; /* 当队列只有一个结点时应防止丢失队尾指针 */
    free(p);
    return OK;
}
```

#### (4) 链队列的撤消

/\* 将链队列Q的队首元素出队 \*/

```
void Destroy_LinkQueue(LinkQueue *Q)
{
    while (Q.front!=NULL)
    {
        Q.rear=Q.front->next; /* 令尾指针指向队列的第一个结点 */
        free(Q.front); /* 每次释放一个结点 */
        /* 第一次是头结点，以后是元素结点 */
        Q.ront=Q.rear;
    }
}
```

220

220

## 队列与广度优先搜索

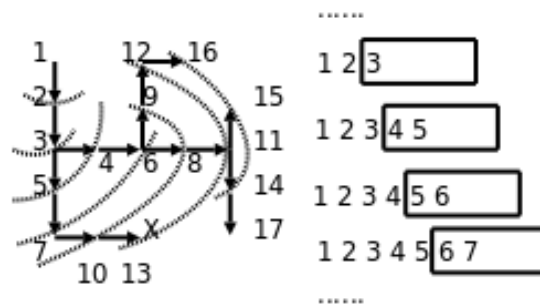
### ■ 用广度优先搜索解迷宫问题

```
将起点标记为已走过并入队;
while (队列非空) {
    出队一个点p;
    if (p这个点是终点)
        break;
    否则沿右、下、左、上四个方向探索相邻的点
    if (和p相邻的点有路可走, 并且还没走过)
        将相邻的点标记为已走过并入队, 它的前趋就是刚出队的p点;
}
if (p点是终点) {
    打印p点的座标;
    while (p点有前趋) {
        p点 = p点的前趋;
        打印p点的座标;
    }
} else
    没有路线可以到达终点;
```

221

221

## 广度优先搜索



广度优先是一种步步为营的策略, 每次都从各个方向探索一步, 将前线推进一步, 图中的虚线就表示这个前线, 队列中的元素总是由前线的点组成的, 可见正是队列先进先出的性质使这个算法具有了**广度优先**的特点。

222

222

专业铸就品质 梦想成就未来

The Specialty Casts Quality,the Dream Gains Future.

## 第六章 串



## 串

在非数值处理、事务处理等问题常涉及到一系列的字符操作。计算机的硬件结构主要是反映数值计算的要求，因此，**字符串的处理比具体数值处理复杂**。本章讨论串的存储结构及几种基本的处理。

## 串的基本概念

**串(字符串)**：是零个或多个字符组成的有限序列。记作： $S = "a_1a_2a_3..."$ ，其中S是串名， $a_i(1 \leq i \leq n)$ 是单个，可以是字母、数字或其它字符。

**串值**：双引号括起来的字符序列是串值。

**串长**：串中所包含的字符个数称为该串的长度。

**空串(空的字符串)**：长度为零的串称为空串，它不包含任何字符。

**空格串(空白串)**：构成串的所有字符都是空格的串称为空白串。

## 串的基本概念

**注意：**空串和空白串的不同，例如 “ ” 和 “ ” 分别表示长度为1的**空白串**和长度为0的**空串**。

**子串(substring)：**串中任意个连续字符组成的子序列称为该串的**子串**，包含子串的串相应地称为**主串**。

**子串的序号：**将子串在主串中首次出现时的该子串的首字符对应在主串中的序号，称为子串在主串中的序号（或位置）。

例如，设有串A和B分别是：

A= “这是字符串” ， B= “是”

则B是A的子串，A为主串。B在A中出现了两次，其中首次出现所对应的主串位置是3。因此，称B在A中的序号为3 。



## 串的基本概念

特别地，**空串是任意串的子串，任意串是其自身的子串。**

**串相等：**如果两个串的串值相等(相同)，称这两个串相等。换言之，只有当两个串的长度相等，且各个对应位置的字符都相同时才相等。

通常在程序中使用的串可分为两种：**串变量和串常量。**

串常量和整常数、实常数一样，在程序中只能被引用但不能不能改变其值，即只能读不能写。通常串常量是由直接量来表示的，例如语句错误(“溢出”)中“溢出”是直接量。

串变量和其它类型的变量一样，其值是可以改变。



## 串的抽象数据类型定义

ADT String{

数据对象:  $D = \{ a_i | a_i \in \text{CharacterSet}, i=1,2,\dots,n, n \geq 0 \}$

数据关系:  $R = \{ \langle a_{i-1}, a_i \rangle | a_{i-1}, a_i \in D, i=2,3,\dots,n \}$

基本操作:

StrAssign(t, chars)

初始条件: chars是一个字符串常量。

操作结果: 生成一个值为chars的串t。

StrConcat(s, t)

初始条件: 串s, t 已存在。

操作结果: 将串t联结到串s后形成新串存放在s中。

StrLength(t)

初始条件: 字符串t已存在。

操作结果: 返回串t中的元素个数, 称为串长。

SubString(s, pos, len, sub)

初始条件: 串s, 已存在,  $1 \leq \text{pos} \leq \text{StrLength}(s)$  且  $0 \leq \text{len} \leq \text{StrLength}(s) - \text{pos} + 1$ 。

操作结果: 用sub返回串s的第pos个字符起长度为len的子串。

.....

} ADT String

229



联航精英训练营

229

## 串的存储表示和实现

串是一种特殊的线性表, 其存储表示和线性表类似, 但又不完全相同。串的存储方式取决于将要串所进行的操作。串在计算机中有3种表示方式:

- ◆ **定长顺序存储表示**: 将串定义成字符数组, 利用串名可以直接访问串值。用这种表示方式, 串的存储空间在编译时确定, 其大小不能改变。
- ◆ **堆分配存储方式**: 仍然用一组地址连续的存储单元来依次存储串中的字符序列, 但串的存储空间是在程序运行时根据串的实际长度动态分配的。
- ◆ **块链存储方式**: 是一种链式存储结构表示。

230



联航精英训练营

230

## 串的定长顺序存储表示

这种存储结构又称为串的顺序存储结构。是用一组连续的存储单元来存放串中的字符序列。所谓定长顺序存储结构，是直接使用定长的字符数组来定义，数组的上界预先确定。

定长顺序存储结构定义为：

```
#define MAX_STRLEN 256
typedef struct
{ char str[MAX_STRLEN];
  int length;
} StringType;
```

231



联航精英训练营

231

## 串的联结操作

```
Status StrConcat ( StringType s, StringType t)
/* 将串t联结到串s之后，结果仍然保存在s中 */
{ int i, j;
  if ((s.length+t.length)>MAX_STRLEN)
    Return ERROR; /* 联结后长度超出范围 */
  for (i=0; i<t.length; i++)
    s.str[s.length+i]=t.str[i]; /* 串t联结到串s之后 */
  s.length=s.length+t.length; /* 修改联结后的串长度 */
  return OK;
}
```

232



联航精英训练营

232

## 求子串操作

```
Status SubString (StringType s, int pos, int len, StringType *sub)
{ int k, j;
  if (pos<1||pos>s.length||len<0||len>(s.length-pos+1))
    return ERROR; /* 参数非法 */
  sub->length=len-pos+1; /* 求得子串长度 */
  for (j=0, k=pos; k<=len; k++, j++)
    sub->str[j]=s.str[i]; /* 逐个字符复制求得子串 */
  return OK;
}
```

233



233

## 串的堆分配存储表示

**实现方法：**系统提供一个空间足够大且地址连续的存储空间(称为“堆”)供串使用。可使用C语言的动态存储分配函数malloc()和free()来管理。

**特点是：**仍然以一组地址连续的存储空间来存储字符串值，但其所需的存储空间是在程序执行过程中动态分配，故是动态的，变长的。

串的堆式存储结构的类型定义

```
typedef struct
{ char *ch; /* 若非空，按长度分配，否则为NULL */
  int length; /* 串的长度 */
} HString;
```

234



234

## 串的联结操作

```
Status Hstring *StrConcat(HString *T, HString *s1, HString *s2)
/* 用T返回由s1和s2联结而成的串 */
{ int k, j, t_len;
  if (T.ch) free(T); /* 释放旧空间 */
  t_len=s1->length+s2->length;
  if ((p=(char *)malloc(sizeof(char)*t_len))==NULL)
    { printf( "系统空间不够, 申请空间失败! \n" );
      return ERROR ; }
  for (j=0; j<s1->length; j++)
    T->ch[j]=s1->ch[j]; /* 将s1复制到串T中 */
  for (k=s1->length, j=0; j<s2->length; k++, j++)
    T->ch[k]=s2->ch[j]; /* 将s2复制到串T中 */
  free(s1->ch);
  free(s2->ch);
  return OK;
}
```

235



235

## 串的链式存储表示

串的链式存储结构和线性表的串的链式存储结构类似, 采用单链表来存储串, 结点的构成是:

- ◆ **data域**: 存放字符, data域可存放的字符个数称为结点的大小;
- ◆ **next域**: 存放指向下一结点的指针。

若每个结点仅存放一个字符, 则结点的指针域就非常多, 造成系统空间浪费, 为节省存储空间, 考虑串结构的特殊性, 使每个结点存放若干个字符, 这种结构称为块链结构。如下图是块大小为3的串的块链式存储结构示意图。

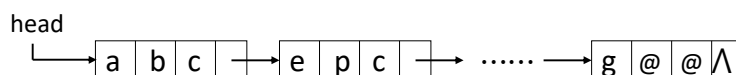


图 串的块链式存储结构示意图

236



236

串的块链式存储的类型定义包括：

**(1) 块结点的类型定义**

```
#define BLOCK_SIZE 4
typedef struct Blstrtype
{ char data[BLOCK_SIZE];
  struct Blstrtype *next;
}BNODE;
```

**(2) 块链串的类型定义**

```
typedef struct
{ BNODE head; /* 头指针 */
  int Strlen; /* 当前长度 */
}Blstring;
```

在这种存储结构下，结点的分配总是完整的结点为单位，因此，为使一个串能存放在整数个结点中，**在串的末尾填上不属于串值的特殊字符**，以表示串的终结。

当一个块(结点)内存放多个字符时，往往会使操作过程变得较为复杂，如在串中插入或删除字符操作时通常需要在块间移动字符。

237

237

## 串的模式匹配算法

**模式匹配(模范匹配)**：子串在主串中的定位称为模式匹配或串匹配(字符串匹配)。模式匹配成功是指在主串S中能够找到模式串T，否则，称模式串T在主串S中不存在。

模式匹配的应用在非常广泛。例如，在文本编辑程序中，我们经常要查找某一特定单词在文本中出现的位置。显然，解此问题的有效算法能极大地提高文本编辑程序的响应性能。

模式匹配是一个较为复杂的串操作过程。迄今为止，人们对串的模式匹配提出了许多思想和效率各不相同的计算机算法。介绍两种主要的模式匹配算法。

238

238

## Brute-Force 模式匹配算法

设S为目标串，T为模式串，且不妨设：

$S = "s_0s_1s_2...s_{n-1}"$  ,  $T = "t_0t_1t_2...t_{m-1}"$

串的匹配实际上是对合法的位置 $0 \leq i \leq n-m$ 依次将目标串中的子串 $s[i...i+m-1]$ 和模式串 $t[0...m-1]$ 进行比较：

- ◆ 若 $s[i...i+m-1] = t[0...m-1]$ ：则称从位置i开始的匹配成功，亦称模式t在目标s中出现；
- ◆ 若 $s[i...i+m-1] \neq t[0...m-1]$ ：从i开始的匹配失败。位置i称为位移，当 $s[i...i+m-1] = t[0...m-1]$ 时，i称为有效位移；当 $s[i...i+m-1] \neq t[0...m-1]$ 时，i称为无效位移。

239



联航精英训练营

239

这样，串匹配问题可简化为找出某给定模式T在给定目标串S中首次出现的有效位移。

### 算法实现

```
int IndexString(StringType s, StringType t, int pos)
/* 采用顺序存储方式存储主串s和模式t, */
/* 若模式t在主串s中从第pos位置开始有匹配的子串, */
/* 返回位置, 否则返回-1 */
{ char *p, *q;
  int k, j;
  k=pos-1; j=0; p=s.str+pos-1; q=t.str;
  /* 初始匹配位置设置 */
  /* 顺序存放时第pos位置的下标值为pos-1 */
  while (k<s.length)&&(j<t.length)
  { if (*p==*q) { p++; q++; k++; j++; }
    else { k=k-j+1; j=0; q=t.str; p=s.str+k; }
    /* 重新设置匹配位置 */
  }
  if (j==t.length)
    return(k-t.length); /* 匹配, 返回位置 */
  else return(-1); /* 不匹配, 返回-1 */
}
```

240

240



该算法简单，易于理解。在一些场合的应用里，如文字处理中的文本编辑，其效率较高。

该算法的时间复杂度为 $O(n*m)$ ，其中 $n$ 、 $m$ 分别是主串和模式串的长度。通常情况下，实际运行过程中，该算法的执行时间近似于 $O(n+m)$ 。

#### 理解该算法的关键点

当第一次 $s_k \neq t_j$ 时：主串要退回到 $k-j+1$ 的位置，而模式串也要退回到第一个字符（即 $j=0$ 的位置）。

比较出现 $s_k \neq t_j$ 时：则应该有 $s_{k-1}=t_{j-1}$ ，...， $s_{k-j+1}=t_1$ ， $s_{k-j}=t_0$ 。

241

241

## 模式匹配的一种改进算法

该改进算法是由D.E.Knuth，J.H.Morris和V.R.Pratt提出来的，简称为KMP算法。其改进在于：

每当一趟匹配过程出现字符不相等时，主串指示器不用回溯，而是利用已经得到的“部分匹配”结果，将模式串的指示器向右“滑动”尽可能远的一段距离后，继续进行比较。

242

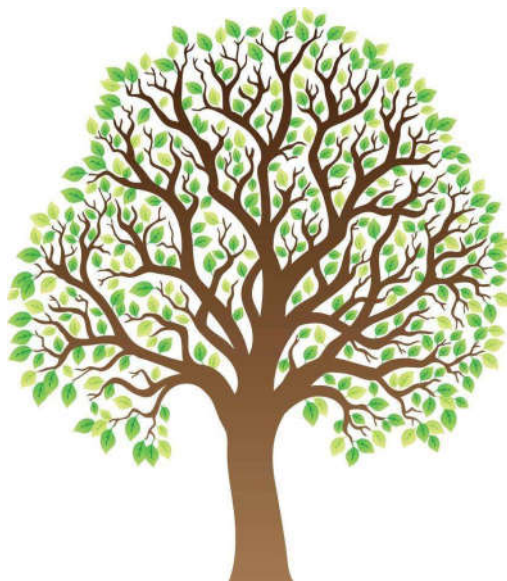
242

专业铸就品质 梦想成就未来

The Specialty Casts Quality,the Dream Gains Future.

## 第七章 树与二叉树

## 树



245



联航精英训练营

245

## 树的定义

树(Tree)是 $n(n \geq 0)$ 个结点的有限集合 $T$ ，若 $n=0$ 时称为空树，否则：

- (1) 有且只有一个特殊的称为树的根(Root)结点；
- (2) 若 $n > 1$ 时，其余的结点被分为 $m(m > 0)$ 个互不相交的子集 $T_1, T_2, T_3, \dots, T_m$ ，其中每个子集本身又是一棵树，称其为根的子树(Subtree)。这是树的递归定义，即用树来定义树，而只有一个结点的树必定仅由根组成，如下图所示。



(a) 只有根结点

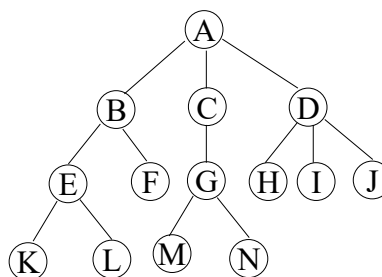


图 树的示例形式

(b) 一般的树

246



联航精英训练营

246

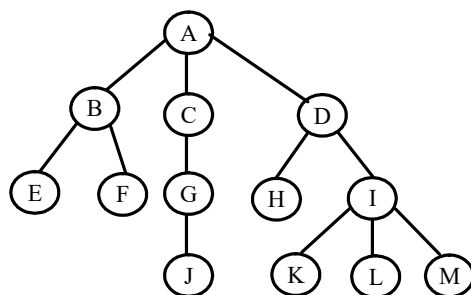
## 树的表示形式

1. 倒悬树（树形表示法）
2. 嵌套集合（文氏图）表示法
3. 广义表形式（括号表示法）
4. 凹入表示法

树的表示方法的多样化说明了树结构的重要性。

## 倒悬树（树形表示法）

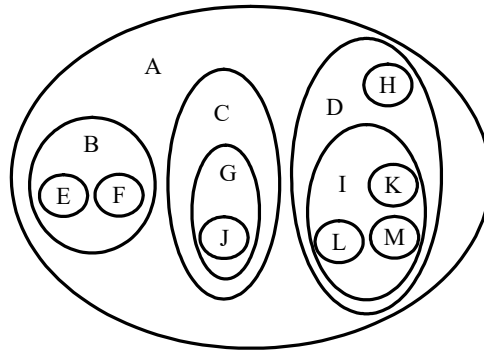
**(1) 倒悬树（树形表示法）。**这是树的最基本的表示,使用一棵倒置的树表示树结构,非常直观和形象。下图就是采用这种表示法。



倒悬树（树形表示法）

## 嵌套集合（文氏图）表示法

(2) **嵌套集合（文氏图）表示法**。使用集合以及集合的包含关系描述树结构，对于任何两个集合，或者不相交。下图就是树的文氏图表示法。



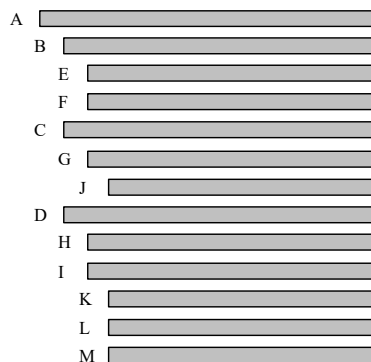
嵌套集合（文氏图）表示法

249

249

## 凹入表示法

(3) **凹入表示法**。使用线段的伸缩描述树结构。下图是树的凹入表示法。



凹入表示法

250

250

## 广义表形式（括号表示法）

(4) **广义表形式（括号表示法）**。将树的根结点写在括号的左边,除根结点之外的其余结点写在括号中并用逗号间隔来描述树结构。下图是树的括号表示法。

**A(B(E,F),C(G(J)),D(H,I(K,L,M)))**

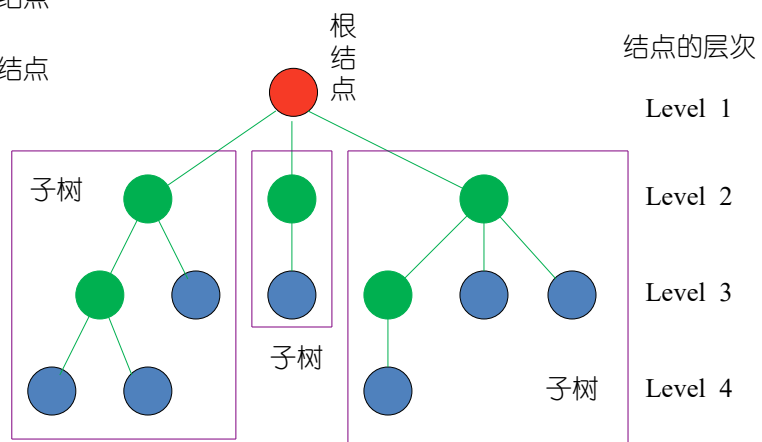
**广义表形式（括号表示法）**

251

251

## 树的示意图

- 根结点
- 叶子结点
- 分支结点



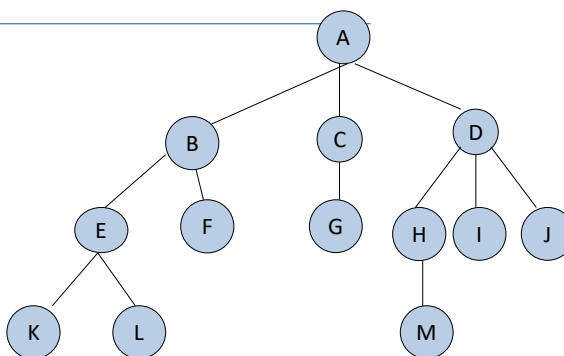
252

252

## 树的实例



(a) 只有根结点的树



(b) 一般的树

(b) 中A为根结点，其余结点分成三个互不相交的子集： $T1=\{B, E, F, K, L\}$ ， $T2=\{C, G\}$ ， $T3=\{D, H, I, J, M\}$ 而 $T1, T2, T3$ 本身又都是只有一个根结点的树。

253

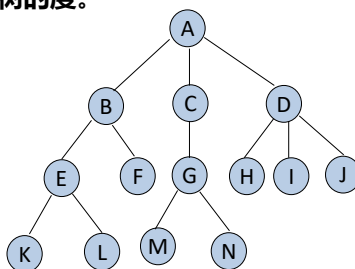
253

## 树的基本术语

- (1) **结点(node)**：一个数据元素及其若干指向其子树的分支。
- (2) **结点的度(degree)、树的度**：结点所拥有的子树的棵数称为**结点的度**。树中结点度的最大值称为**树的度**。



(a) 只有根结点



树的示例形式

(b) 一般的树

如图中结点A的度是3，结点B的度是2，结点M的度是0，树的度是3。

254

254

## 树的基本术语

### (3) 叶子(left)结点、非叶子结点

树中度为0的结点称为**叶子结点**(或终端结点)。相应地,度不为0的结点称为**非叶子结点**(或非终端结点或分支结点)。除根结点外,分支结点又称为**内部结点**。

如上图(b)中结点H、I、J、K、L、M、N是叶子结点,而所有其它结点都是分支结点。

### (4) 孩子结点、双亲结点、兄弟结点

一个结点的子树的根称为该结点的**孩子结点**(child)或子结点;相应地,该结点是其孩子结点的**双亲结点**(parent)或父结点。

如图(b)中结点B、C、D是结点A的子结点,而结点A是结点B、C、D的父结点;类似地结点E、F是结点B的子结点,结点B是结点E、F的父结点。

同一双亲结点的所有子结点互称为**兄弟结点**。

如图(b)中结点B、C、D是兄弟结点;结点E、F是兄弟结点。

### (5) 层次、堂兄弟结点

规定树中根结点的层次为1,其余结点的层次等于其双亲结点的层次加1。

若某结点在第 $l(l \geq 1)$ 层,则其子结点在第 $l+1$ 层。

双亲结点在同一层上的所有结点互称为**堂兄弟结点**。如图(b)中结点E、F、G、H、I、J。

255



联航精英训练营

255

## 树的基本术语

### (6) 结点的层次路径、祖先、子孙

从根结点开始,到达某结点p所经过的所有结点成为结点p的**层次路径**(有且只有一条)。

结点p的层次路径上的所有结点(p除外)称为p的**祖先**(ancestor)。

以某一结点为根的子树中的任意结点称为该结点的**子孙结点**(descent)。

(7) **树的深度(depth)**: 树中结点的最大层次值,又称为树的高度,如图(b)中树的高度为4。

(8) **有序树和无序树**: 对于一棵树,若其中每一个结点的子树(若有)具有一定的次序,则该树称为**有序树**,否则称为**无序树**。

(9) **森林(forest)**: 是 $m(m \geq 0)$ 棵互不相交的树的集合。显然,若将一棵树的根结点删除,剩余的子树就构成了森林。

256



联航精英训练营

256



## 树的抽象数据类型定义

```
ADT Tree{  
    数据对象D: D是具有相同数据类型的数据元素的集合。  
    数据关系R: 若D为空集, 则称为空树;  
    .....  
    基本操作:  
    .....  
} ADT Tree
```

257



联航精英训练营

257

## 树的基本操作

- 1) InitTree ( &T );                   //构造空树 T。
- 2) DestroyTree ( &T );               //销毁树 T。
- 3) CreateTree ( &T, definition );    //按 definition 构造树 T。
- 4) ClearTree ( &T );                 //将树 T 清空。
- 5) TreeEmpty ( T );                  //若树 T 为空, 返回 TURE, 否则返回 FALSE。
- 6) TreeDepth ( T );                  //返回树 T 的深度。
- 7) Root ( T );                        //返回 T 的根结点。
- 8) Value ( T, &cur\_e );              //返回 T 树中 cur\_e 结点的值。
- 9) Assign ( T, cur\_e, value );       //将 T 树中结点 cur\_e 的值赋值为 value。
- 10) Parent ( T, cur\_e );             //返回 T 树 cur\_e 结点的双亲。
- 11) LeftChild ( T, cur\_e );         //返回 T 树 cur\_e 结点的最左孩子。
- 12) RightSibling ( T, cur\_e );       //返回 T 树 cur\_e 结点的右兄弟。
- 13) InsertChild ( &T, &p, i, c );    //将 c 插入到树 T 中 p 所指向的第 i 棵子树中。
- 14) DeleteChild ( &T, &p, i );       //删除树 T 中 p 所指向的第 i 棵子树。
- 15) TraverseTree ( T, Visit( ) );  
按某种次序对 T 树的每个结点调用函数 Visit( ) 一次且至多一次。也称为按照某种次序对树进行遍历。

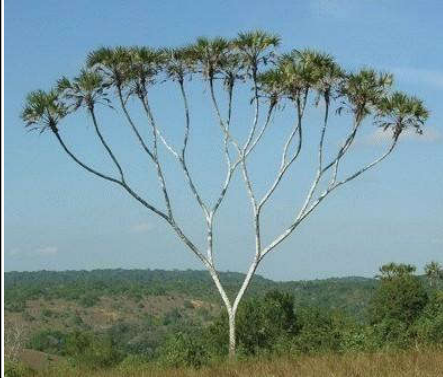
258



联航精英训练营

258

## 二叉树



259

259

## 二叉树的定义

- 二叉树(Binary tree)是 $n(n \geq 0)$ 个结点的有限集合。若 $n=0$ 时称为空树，否则：
  - (1) 有且只有一个特殊的称为树的根(Root)结点；
  - (2) 若 $n > 1$ 时，其余的结点被分成为二个互不相交的子集 $T_1, T_2$ ，分别称之为左、右子树，并且左、右子树又都是二叉树。由此可知，二叉树的定义是递归的。

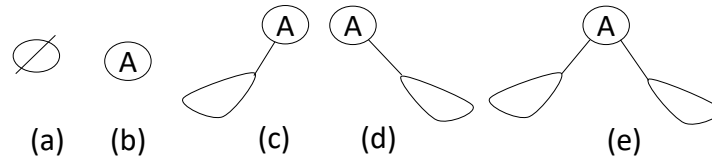
二叉树在树结构中起着非常重要的作用。因为二叉树结构简单，存储效率高，树的操作算法相对简单，且任何树都很容易转化成二叉树结构。上节中引入的有关树的术语也都适用于二叉树。

260

260

## 二叉树的基本形态

### ■ 二叉树有5种基本形态



(a) 空二叉树 (b) 单结点二叉树 (c) 右子树为空  
(d) 左子树为空 (e) 左、右子树都不空  
二叉树的基本形态

## 二叉树的性质

### ■ 性质1：在非空二叉树中，第 $i$ 层上至多有 $2^{i-1}$ 个结点( $i \geq 1$ )。

证明：用数学归纳法证明。

当 $i=1$ 时：只有一个根结点， $2^{1-1}=2^0=1$ ，命题成立。

现假设对 $i > 1$ 时，处在第 $i-1$ 层上至多有 $2^{(i-1)-1}$ 个结点。

由归纳假设知，第 $i-1$ 层上至多有 $2^{i-2}$ 个结点。由于二叉树每个结点的度最大为2，故在第 $i$ 层上最大结点数为第 $i-1$ 层上最大结点数的2倍。

$$\text{即 } 2 \times 2^{i-2} = 2^{i-1}$$

证毕

## 二叉树的性质

**性质2：**深度为k的二叉树至多有 $2^k-1$ 个结点 ( $k \geq 1$ )。

**证明：**深度为k的二叉树的最大的结点数为二叉树中每层上的最大结点数之和。

由性质1知，二叉树的第一层、第二层、第k层上的结点数至多有： $2^0, 2^1, \dots, 2^{k-1}$ 。

$\therefore$  总的结点数至多有： $2^0 + 2^1 + \dots + 2^{k-1} = 2^k - 1$  证毕

**性质3：**对任何一棵二叉树，若其叶子结点数为 $n_0$ ，度为2的结点数为 $n_2$ ，则 $n_0 = n_2 + 1$ 。

**证明：**设二叉树中度为1的结点数为 $n_1$ ，二叉树中总结点数为N，因为二叉树中所有结点的度均小于或等于2，则有： $N = n_0 + n_1 + n_2$

再看二叉树中的分支数：

除根结点外，其余每个结点都有唯一的一个进入分支，而所有这些分支都是由度为1和2的结点射出的。设B为二叉树中的分支总数，则有： $N = B + 1$

$$\therefore B = n_1 + 2 \times n_2$$

$$\therefore N = B + 1 = n_1 + 2 \times n_2 + 1$$

$$\therefore n_0 + n_1 + n_2 = n_1 + 2 \times n_2 + 1$$

$$\text{即 } n_0 = n_2 + 1$$

证毕

263

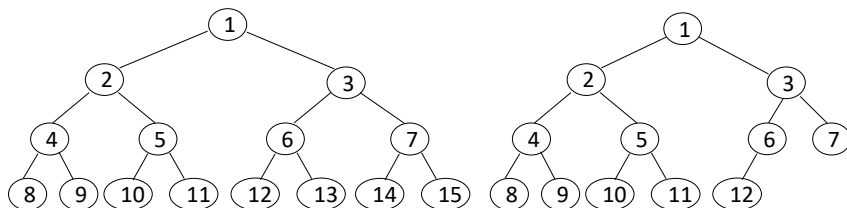
联航精英训练营

263

## 满二叉树与完全二叉树

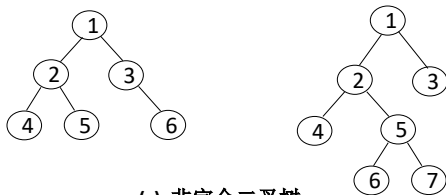
**满二叉树** 一棵深度为k且有 $2^k-1$ 个结点的二叉树称为**满二叉树**(Full Binary Tree)。

**完全二叉树**(Complete Binary Tree)：如果深度为k，由n个结点的二叉树，当且仅当其每一个结点都与深度为k的满二叉树中编号从1到n的结点一一对应，该二叉树称为完全二叉树。



(a) 满二叉树

(b) 完全二叉树



(c) 非完全二叉树

特殊形态的二叉树

264

联航精英训练营

264

## 满二叉树与完全二叉树

### 满二叉树的特点:

- ◆基本特点是每一层上的结点数总是最大结点数。
- ◆满二叉树的所有的支结点都有左、右子树。
- ◆可对满二叉树的结点进行连续编号,若规定从根结点开始,按“**自上而下、自左至右**”的原则进行。

**完全二叉树(Complete Binary Tree):** 如果深度为 $k$ ,由 $n$ 个结点的二叉树,当且仅当其每一个结点都与深度为 $k$ 的满二叉树中编号从1到 $n$ 的结点——对应,该二叉树称为完全二叉树。

或深度为 $k$ 的满二叉树中编号从1到 $n$ 的前 $n$ 个结点构成了一棵深度为 $k$ 的完全二叉树。

其中  $2^{k-1} \leq n \leq 2^k - 1$ 。

**完全二叉树是满二叉树的一部分,而满二叉树是完全二叉树的特例。**

265



联航精英训练营

265

### 完全二叉树的特点:

若完全二叉树的深度为 $k$ ,则所有的叶子结点都出现在第 $k$ 层或 $k-1$ 层。对于任一结点,如果其右子树的最大层次为 $l$ ,则其左子树的最大层次为 $l$ 或 $l+1$ 。

**性质4:**  $n$ 个结点的完全二叉树深度为:  $\lfloor \log_2 n \rfloor + 1$ 。

其中符号:  $\lfloor x \rfloor$  表示不大于 $x$ 的最大整数。

$\lceil x \rceil$  表示不小于 $x$ 的最小整数。

**证明:** 假设完全二叉树的深度为 $k$ ,则根据性质2及完全二叉树的定义有:

$$2^{k-1} - 1 < n \leq 2^k - 1 \quad \text{或} \quad 2^{k-1} \leq n < 2^k$$

取对数得:  $k - 1 < \log_2 n < k$  因为 $k$ 是整数。

$$\therefore k = \lfloor \log_2 n \rfloor + 1 \quad \text{证毕}$$

266

266

- **性质5**：若对一棵有 $n$ 个结点的完全二叉树(深度为 $\lfloor \log_2 n \rfloor + 1$ )的结点按层(从第1层到第 $\lfloor \log_2 n \rfloor + 1$ 层)序自左至右进行编号,则对于编号为 $i$  ( $1 \leq i \leq n$ )的结点:

- (1) 若 $i=1$ : 则结点 $i$ 是二叉树的根,无双亲结点; 若 $i>1$ , 则其双亲结点编号是 $\lfloor i/2 \rfloor$ 。
- (2) 如果 $2i>n$ : 则结点 $i$ 为叶子结点, 无左孩子; 否则, 其左孩子结点编号是 $2i$ 。
- (3) 如果 $2i+1>n$ : 则结点 $i$ 无右孩子; 否则, 其右孩子结点编号是 $2i+1$ 。

**证明**: 用数学归纳法证明。首先证明(2)和(3), 然后由(2)和(3)导出(1)。

当 $i=1$ 时, 由完全二叉树的定义知, 结点 $i$ 的左孩子的编号是2, 右孩子的编号是3。

若 $2>n$ , 则二叉树中不存在编号为2的结点, 说明**结点 $i$ 的左孩子**不存在。

若 $3>n$ , 则二叉树中不存在编号为3的结点, 说明**结点 $i$ 的右孩子**不存在。

现假设对于编号为 $j$  ( $1 \leq j \leq i$ )的结点, (2)和(3)成立。即:

- ◆ 当 $2j \leq n$ : 结点 $j$ 的左孩子编号是 $2j$ ; 当 $2j > n$ 时, 结点 $j$ 的左孩子结点不存在。
- ◆ 当 $2j+1 \leq n$ : 结点 $j$ 的右孩子编号是 $2j+1$ ; 当 $2j+1 > n$ 时, 结点 $j$ 的右孩子结点不存在。

当 $i=j+1$ 时, 由完全二叉树的定义知, 若结点 $i$ 的左孩子结点存在, 则其左孩子结点的编号一定等于编号为 $j$ 的右孩子的编号加1, 即结点 $i$ 的左孩子的编号为:

$$(2j+1)+1=2(j+1)=2i$$

如下图所示, 且有 $2i \leq n$ 。相反, 若 $2i > n$ , 则左孩子结点不存在。同样地, 若结点 $i$ 的右孩子结点存在, 则其右孩子的编号为:  $2i+1$ , 且有 $2i+1 \leq n$ 。相反, 若 $2i+1 > n$ , 则左孩子结点不存在。结论(2)和(3)得证。

再由(2)和(3)来证明(1)。

当 $i=1$ 时, 显然编号为1的是根结点, 无双亲结点。

当 $i>1$ 时, 设编号为 $i$ 的结点的双亲结点的编号为 $m$ , 若编号为 $i$ 的结点是其双亲结点的左孩子, 则由(2)有:

$$i=2m, \text{ 即 } m=\lfloor i/2 \rfloor;$$

若编号为 $i$ 的结点是其双亲结点的右孩子, 则由(3)有:

$$i=2m+1, \text{ 即 } m=\lfloor (i-1)/2 \rfloor;$$

∴ 当 $i>1$ 时, 其双亲结点的编号为 $\lfloor i/2 \rfloor$ 。 证毕

## 完全二叉树中结点*i*和*i*+1的左右孩子

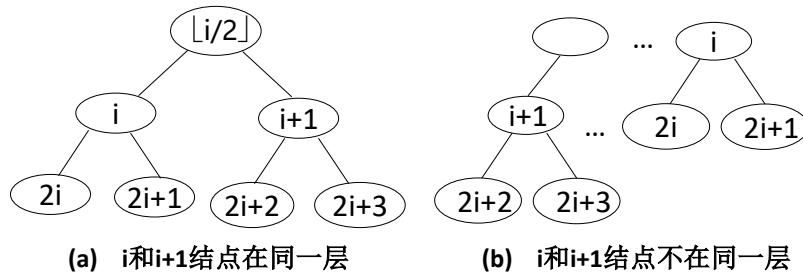


图 完全二叉树中结点*i*和*i*+1的左右孩子

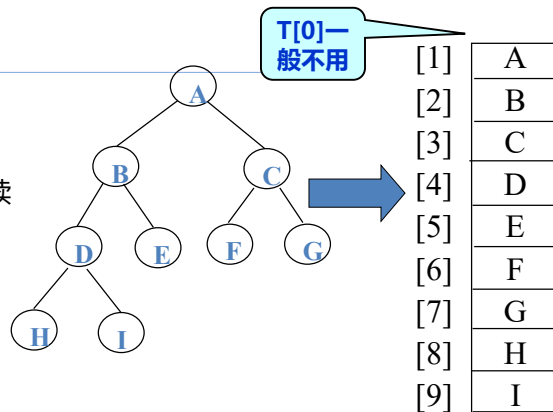
269

269

## 二叉树的存储结构

### 一、顺序存储结构

按二叉树的结点“自上而下、从左至右”编号，用一组连续的存储单元存储。



问：顺序存储后能否复原成唯一对应的二叉树形状？

答：若是完全/满二叉树则可以做到唯一复原。

而且有规律：下标值为*i*的双亲，其左孩子的下标值必为2*i*，其右孩子的下标值必为2*i* + 1 例如，对应[2]的两个孩子必为[4]和[5]，即B的左孩子必是D，右孩子必为E。

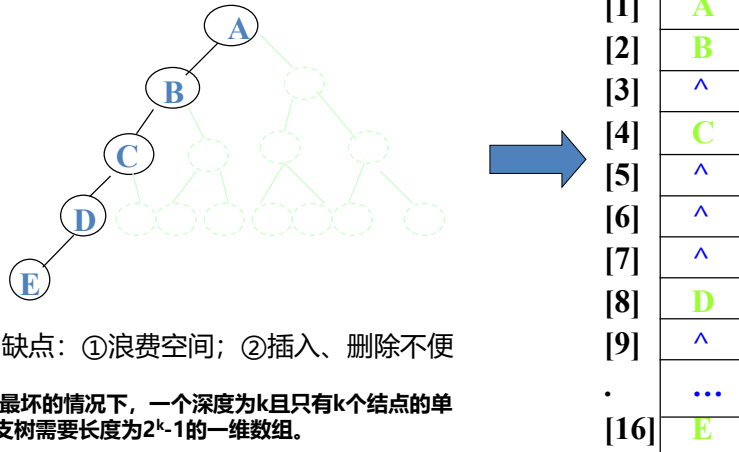
270

270

## 讨论：不是完全二叉树怎么办？

答：一律转为完全二叉树！

方法很简单，将各层空缺处统统补上“虚结点”，其内容为空。

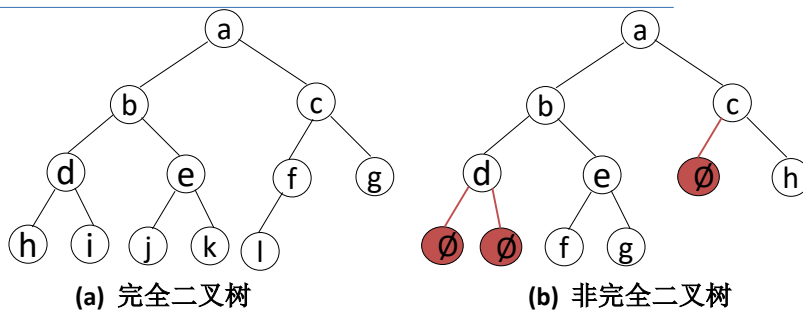


271

联航精英训练营

271

## 二叉树及其顺序存储形式



1	2	3	4	5	6	7	8	9	10	11	12
a	b	c	d	e	f	g	h	i	j	k	l

(c) 完全二叉树的顺序存储形式

1	2	3	4	5	6	7	8	9	10	11
a	b	c	d	e	∅	h	∅	∅	f	g

(d) 非完全二叉树的顺序存储形式

联航精英训练营

272



## 链式存储结构

设计不同的结点结构可构成不同的链式存储结构。

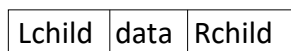
### (1) 结点的类型及其定义

① **二叉链表结点**。有三个域：一个数据域，两个分别指向左右子结点的指针域，如图(a)所示。

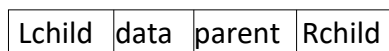
```
typedef struct BTNode
{
    ElemType data;
    struct BTNode *Lchild, *Rchild;
}BTNode;
```

② **三叉链表结点**。除二叉链表的三个域外，再增加一个指针域，用来指向结点的父结点，如图(b)所示。

```
typedef struct BTNode_3
{
    ElemType data;
    struct BTNode_3 *Lchild, *Rchild, *parent;
}BTNode_3;
```



(a) 二叉链表结点



(b) 三叉链表结点

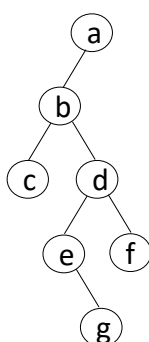
图 链表结点结构形式

273

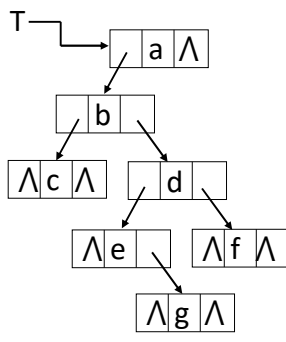
273

## 二叉树的链式存储形式

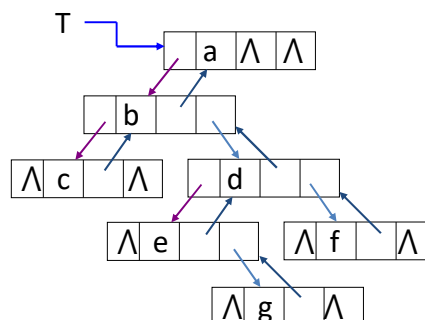
例有一棵一般的二叉树，如图(a)所示。以二叉链表和三叉链表方式存储的结构图分别如图(b)、(c)所示。



(a) 二叉树



(b) 二叉链表



(c) 三叉链表

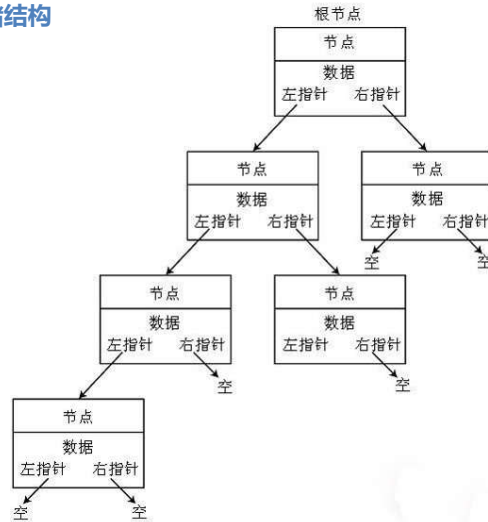
图 二叉树及其链式存储结构

274

274

## 二叉树的存储结构

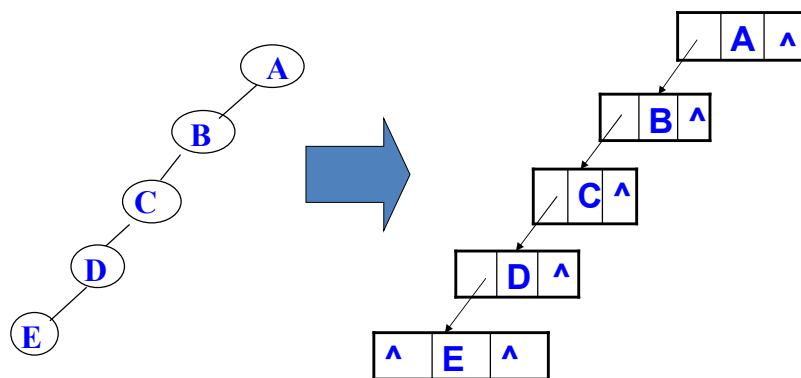
### 二、链式存储结构



275

275

## 二叉树链式存储结构



优点：①不浪费空间；②插入、删除方便

276

276

## 遍历二叉树

**遍历二叉树**(Traversing Binary Tree): 是指**按指定的规律**对二叉树中的**每个结点访问一次且仅访问一次**。

所谓**访问**是指对结点做某种处理。如: 输出信息、修改结点的值等。

二叉树是一种非线性结构, 每个结点都可能由左、右两棵子树, 因此, 需要寻找一种规律, 使二叉树上的结点能排列在一个线性队列上, 从而便于遍历。

二叉树的基本组成: 根结点、左子树、右子树。若能依次遍历这三部分, 就是遍历了二叉树。

## 遍历二叉树

若以**L**、**D**、**R**分别表示**遍历左子树**、**遍历根结点**和**遍历右子树**, 则有六种遍历方案: DLR、LDR、LRD、DRL、RDL、RLD。若规定先左后右, 则只有前三种情况三种情况, 分别是:

- ◆ DLR——先(根)序遍历。
- ◆ LDR——中(根)序遍历。
- ◆ LRD——后(根)序遍历。

对于二叉树的遍历, 分别讨论**递归遍历算法**和**非递归遍历算法**。递归遍历算法具有非常清晰的结构, 但初学者往往难以接受或怀疑, 不敢使用。实际上, 递归算法是由系统通过使用堆栈来实现控制的。而非递归算法中的控制是由设计者定义和使用堆栈来实现的。

## 先序遍历二叉树-递归算法

### 1 递归算法

算法的递归定义是：

若二叉树为空，则遍历结束；否则

- (1) 访问根结点；
- (2) 先序遍历左子树(递归调用本算法)；
- (3) 先序遍历右子树(递归调用本算法)。

### 先序遍历的递归算法

```
void PreorderTraverse(BTNode *T)
{ if (T!=NULL)
    { visit(T->data); /* 访问根结点 */
      PreorderTraverse(T->Lchild);
      PreorderTraverse(T->Rchild);
    }
}
```

**说明：**visit()函数是访问结点的数据域，其要求视具体问题而定。树采用二叉链表的存储结构，用指针变量T来指向。

279



279

## 先序遍历二叉树-迭代算法

设T是指向二叉树根结点的指针变量，**非递归算法**是：

若二叉树为空，则返回；否则，令p=T；

- (1) 访问p所指向的结点；
- (2) q=p->Rchild，若q不为空，则q进栈；
- (3) p=p->Lchild，若p不为空，转(1)，否则转(4)；
- (4) 退栈到p，转(1)，直到栈空为止。

### 算法实现：

```
#define MAX_NODE 50
void PreorderTraverse( BTNode *T)
{
    BTNode *Stack[MAX_NODE], *p=T, *q;
    int top=0;
    if (T==NULL) printf(" Binary Tree is Empty!\n");
    else { do
        { visit( p-> data ); q=p->Rchild;
          if ( q!=NULL ) stack[++top]=q;
          p=p->Lchild;
          if (p==NULL) { p=stack[top]; top--; }
        }
        while (p!=NULL);
    }
}
```

280



280

## 中序遍历二叉树-递归算法

算法的递归定义是：

若二叉树为空，则遍历结束；否则

- (1) 中序遍历左子树(递归调用本算法)；
- (2) 访问根结点；
- (3) 中序遍历右子树(递归调用本算法)。

### 中序遍历的递归算法

```
void InorderTraverse(BTNode *T)
{ if (T!=NULL)
    { InorderTraverse(T->Lchild);
      visit(T->data); /* 访问根结点 */
      InorderTraverse(T->Rchild);
    }
}
```

281



联航精英训练营

281

## 中序遍历二叉树-非递归算法

设T是指向二叉树根结点的指针变量，非递归算法是：

若二叉树为空，则返回；否则，令p=T

- (1) 若p不为空，p进栈，p=p->Lchild；
  - (2) 否则(即p为空)，退栈到p，访问p所指向的结点；
  - (3) p=p->Rchild，转(1)；
- 直到栈空为止。

### 算法实现：

```
#define MAX_NODE 50
void InorderTraverse( BTNode *T)
{ BTNode *Stack[MAX_NODE], *p=T;
  int top=0, bool=1;
  if (T==NULL) printf( " Binary Tree is Empty!\n" );
  else { do
        { while (p!=NULL)
            { stack[++top]=p; p=p->Lchild; }
          if (top==0) bool=0;
          else { p=stack[top]; top--;
                visit( p->data ); p=p->Rchild; }
        } while (bool!=0);
  }
```

282



联航精英训练营

282

## 后序遍历二叉树-递归算法

算法的递归定义是：

若二叉树为空，则遍历结束；否则

- (1) 后序遍历左子树(递归调用本算法)；
- (2) 后序遍历右子树(递归调用本算法)；
- (3) 访问根结点。

后序遍历的递归算法

```
void PostorderTraverse(BTNode *T)
{ if (T!=NULL)
  { PostorderTraverse(T->Lchild);
    PostorderTraverse(T->Rchild);
    visit(T->data); /* 访问根结点 */
  }
}
```

遍历二叉树的算法中基本操作是访问结点，因此，无论是哪种次序的遍历，对有n个结点的二叉树，其时间复杂度均为O(n)。

283



283

如图所示的二叉树表示表达式：(a+b\*(c-d)-e/f)

按不同的次序遍历此二叉树，将访问的结点按先后次序排列起来的次序是：

其先序序列为： -+a\*b-cd/ef

其中序序列为： a+b\*c-d-e/f

其后序序列为： abcd-\*+ef/-

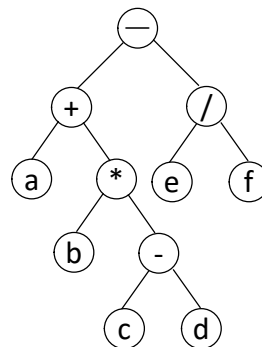


图 表达式 (a+b\*(c-d)-e/f) 二叉树

284

284

## 后序遍历二叉树-非递归算法

在后序遍历中，根结点是最后被访问的。因此，在遍历过程中，当搜索指针指向某一根结点时，不能立即访问，而要先遍历其左子树，此时**根结点进栈**。当其左子树遍历完后，再搜索到该根结点时，还是不能访问，还需遍历其右子树。所以，此**根结点还需再次进栈**，当其右子树遍历完后，再退栈到该根结点时，才能被访问。

因此，设立一个状态标志变量tag：

$$\text{tag} = \begin{cases} 0: & \text{结点暂不能访问} \\ 1: & \text{结点可以被访问} \end{cases}$$

其次，设两个堆栈 $S_1$ 、 $S_2$ ， $S_1$ 保存结点， $S_2$ 保存结点的状态标志变量tag。 $S_1$ 和 $S_2$ 共用一个栈顶指针。

设T是指向根结点的指针变量，非递归算法是：

若二叉树为空，则返回；否则，令 $p=T$ ；

(1) 第一次经过根结点p，不访问：

p进栈 $S_1$ ，tag赋值0，进栈 $S_2$ ， $p=p \rightarrow \text{Lchild}$ 。

(2) 若p不为空，转(1)，否则，取状态标志值tag：

(3) 若tag=0：对栈 $S_1$ ，不访问，不出栈；修改 $S_2$ 栈顶元素值(tag赋值1)，取 $S_1$ 栈顶元素的右子树，即 $p=S_1[\text{top}] \rightarrow \text{Rchild}$ ，转(1)；

(4) 若tag=1： $S_1$ 退栈，访问该结点；

直到栈空为止。

285



285

### 算法实现：

```
#define MAX_NODE 50
void PostorderTraverse( BTreeNode *T)
{ BTreeNode *S1[MAX_NODE], *p=T;
  int S2[MAX_NODE], top=0, bool=1;
  if (T==NULL) printf( "Binary Tree is Empty!\n" );
  else { do
    { while (p!=NULL)
      { S1[++top]=p; S2[top]=0;
        p=p->Lchild;
      }
      if (top==0) bool=0;
      else if (S2[top]==0)
        { p=S1[top]->Rchild; S2[top]=1; }
      else
        { p=S1[top]; top--;
          visit( p->data ); p=NULL;
          /* 使循环继续进行而不至于死循环 */
        }
    } while (bool!=0);
  }
```

286

286

## 层次遍历二叉树

层次遍历二叉树，是从根结点开始遍历，按层次次序“自上而下，从左至右”访问树中的各结点。

为保证是按层次遍历，必须设置一个队列，初始化时空。

设T是指向根结点的指针变量，层次遍历非递归算法是：

若二叉树为空，则返回；否则，令p=T，p入队；

- (1) 队首元素出队到p；
- (2) 访问p所指向的结点；
- (3) 将p所指向的结点的左、右子结点依次入队。直到队空为止。



```
#define MAX_NODE 50
void LevelorderTraverse( BTreeNode *T)
{ BTreeNode *Queue[MAX_NODE], *p=T;
  int front=0, rear=0;
  if (p!=NULL)
  { Queue[++rear]=p; /* 根结点入队 */
    while (front<rear)
    { p=Queue[++front]; visit( p->data );
      if (p->Lchild!=NULL)
        Queue[++rear]=p->Lchild; /* 左结点入队 */
      if (p->Rchild!=NULL)
        Queue[++rear]=p->Rchild; /* 右结点入队 */
    }
  }
}
```



## 求二叉树的叶子结点数

可以直接利用先序遍历二叉树算法求二叉树的叶子结点数。只要将先序遍历二叉树算法中vist()函数简单地进行修改就可以。

**算法实现:**

```
#define MAX_NODE 50
int search_leaves( BTreeNode *T)
{ BTreeNode *Stack[MAX_NODE], *p=T;
  int top=0, num=0;
  if (T!=NULL)
  { stack[++top]=p;
    while (top>0)
    { p=stack[top--];
      if (p->Lchild==NULL&&p->Rchild==NULL) num++;
      if (p->Rchild!=NULL)
        stack[++top]=p->Rchild;
      if (p->Lchild!=NULL)
        stack[++top]=p->Lchild;
    }
  }
  return(num);
}
```

289



联航精英训练营

289

## 求二叉树的深度

利用层次遍历算法可以直接求得二叉树的深度。

**算法实现:**

```
#define MAX_NODE 50
int search_depth( BTreeNode *T)
{ BTreeNode *Stack[MAX_NODE], *p=T;
  int front=0, rear=0, depth=0, level;
  /* level总是指向访问层的最后一个结点在队列的位置 */

  if (T!=NULL)
  { Queue[++rear]=p; /* 根结点入队 */
    level=rear; /* 根是第1层的最后一个结点 */
    while (front<rear)
    { p=Queue[++front];
      if (p->Lchild!=NULL)
        Queue[++rear]=p->Lchild; /* 左结点入队 */
      if (p->Rchild!=NULL)
        Queue[++rear]=p->Rchild; /* 右结点入队 */
      if (front==level)
        /* 正访问的是当前层的最后一个结点 */
        { depth++; level=rear; }
    }
  }
}
```

290



联航精英训练营

290

## 二叉排序树(BST)的定义

二叉排序树(Binary Sort Tree或Binary Search Tree) 的定义为:  
二叉排序树或者是空树, 或者是满足下列性质的二叉树。

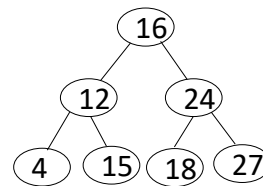
- (1): 若左子树不为空, 则左子树上所有结点的值(关键字)都小于根结点的值;
- (2): 若右子树不为空, 则右子树上所有结点的值(关键字)都大于根结点的值;
- (3): 左、右子树都分别是二叉排序树。

**结论:** 若按中序遍历一棵二叉排序树, 所得到的结点序列是一个递增序列。

BST仍然可以用二叉链表来存储, 如图所示。

结点类型定义如下:

```
typedef struct Node
{
    KeyType key; /* 关键字域 */
    ... /* 其它数据域 */
    struct Node *Lchild, *Rchild;
}BSTNode;
```



二叉排序树

## BST树的查找

### 1 查找思想

首先将给定的K值与二叉排序树的根结点的关键字进行比较: 若相等:  
则查找成功;

- ① 给定的K值小于BST的根结点的关键字: 继续在该结点的左子树上进行查找;
- ② 给定的K值大于BST的根结点的关键字: 继续在该结点的右子树上进行查找。

## BST树的插入

在BST树中插入一个新结点，要保证插入后仍满足BST的性质。

### 1 插入思想

在BST树中插入一个新结点x时，若BST树为空，则令新结点x为插入后BST树的根结点；否则，将结点x的关键字与根结点T的关键字进行比较：

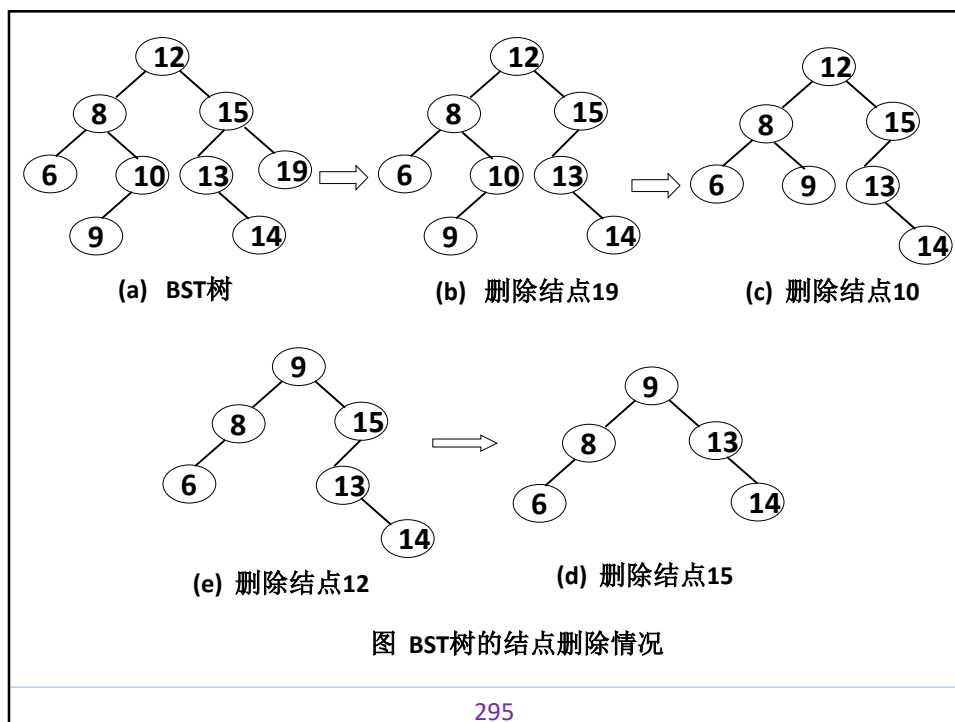
- ① 若相等：不需要插入；
- ② 若 $x.key < T \rightarrow key$ ：结点x插入到T的左子树中；
- ③ 若 $x.key > T \rightarrow key$ ：结点x插入到T的右子树中。

## BST树的删除

### 删除操作过程分析

从BST树上删除一个结点，仍然要保证删除后满足BST的性质。设被删除结点为p，其父结点为f，删除情况如下：

- ① 若p是叶子结点：直接删除p，如图所示。
- ② 若p只有一棵子树(左子树或右子树)：直接用p的左子树(或右子树)取代p的位置而成为f的一棵子树。即原来p是f的左子树，则p的子树成为f的左子树；原来p是f的右子树，则p的子树成为f的右子树，如图(c)、(d)所示。
- ③ 若p既有左子树又有右子树：处理方法有以下两种，可以任选其中一种。
  - ◆ 用p的直接前驱结点代替p。即从p的左子树中选择值最大的结点s放在p的位置(用结点s的内容替换结点p内容)，然后删除结点s。s是p的左子树中的最右边的结点且没有右子树，对s的删除同②，如图(e)所示。
  - ◆ 用p的直接后继结点代替p。即从p的右子树中选择值最小的结点s放在p的位置(用结点s的内容替换结点p内容)，然后删除结点s。s是p的右子树中的最左边的结点且没有左子树，对s的删除同②，如图(f)所示。



295

295

## 性能分析

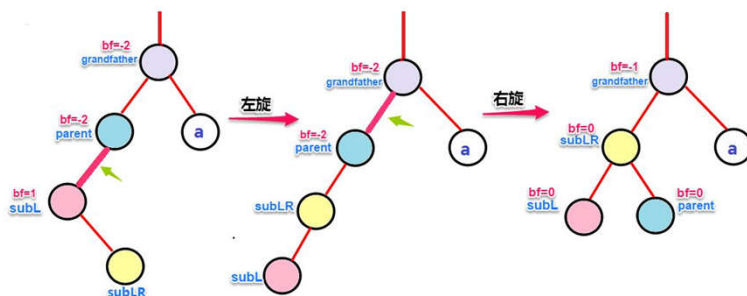
- 每个结点的 $C(i)$ 为该结点的层次数。
- 最坏情况下，当先后插入的关键字有序时，构成的二叉排序树蜕变为单支树，树的深度为其平均查找长度 $(n+1)/2$  (和顺序查找相同)，最好的情况是二叉排序树的形态和折半查找的判定树相同，其平均查找长度和 $\log_2(n)$ 成正比。

296

296

## AVL树（平衡二叉树）

AVL树是最先发明的**自平衡二叉查找树**。在AVL树中任何节点的两个子树的高度最大差别为一，所以它也被称为高度平衡树。**查找、插入和删除在平均和最坏情况下都是 $O(\log n)$** 。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。AVL树得名于它的发明者 G.M. Adelson-Velsky 和 E.M. Landis，他们在 1962 年的论文 "An algorithm for the organization of information" 中发表了它。



297

297

## 红黑树

- **红黑树 (Red Black Tree)** 是一种**自平衡二叉查找树**，是在计算机科学中用到的一种数据结构，典型的用途是实现关联数组。
- 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 Sedgwick 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：它可以在 $O(\log n)$ 时间内做查找，插入和删除，这里的 $n$ 是树中元素的数目。
- 它的统计性能要好于平衡二叉树 (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)。

298

298

## 线索树

遍历二叉树是按一定的规则将树中的结点排列成一个线性序列，即是对非线性结构的线性化操作。如何找到**遍历过程中动态得到**的每个结点的直接前驱和直接后继(第一个和最后一个除外)?如何保存这些信息?

设一棵二叉树有 $n$ 个结点，则有 $n-1$ 条边(指针连线)，而 $n$ 个结点共有 $2n$ 个指针域(Lchild和Rchild)，显然有 **$n+1$ 个空闲指针域**未用。则可以利用这些空闲的指针域来存放结点的直接前驱和直接后继信息。

对结点的指针域做如下规定：

- ◆ 若结点有左孩子，则Lchild指向其左孩子，否则，指向其直接前驱；
- ◆ 若结点有右孩子，则Rchild指向其右孩子，否则，指向其直接后继；

为避免混淆,对结点结构加以改进，增加两个标志域，如下图所示。

299



联航精英训练营

299

Lchild	Ltag	data	Rchild	Rtag
--------	------	------	--------	------

图 线索二叉树的结点结构

$$Ltag = \begin{cases} 0: & \text{Lchild域指示结点的左孩子} \\ 1: & \text{Lchild域指示结点的前驱} \end{cases}$$

$$Rtag = \begin{cases} 0: & \text{Rchild域指示结点的右孩子} \\ 1: & \text{Rchild域指示结点的后继} \end{cases}$$

用这种结点结构构成的二叉树的存储结构；叫做**线索链表**；指向结点前驱和后继的指针叫做**线索**；按照某种次序遍历，加上线索的二叉树称之为**线索二叉树**。

300

300

## 线索二叉树的结点结构与示例

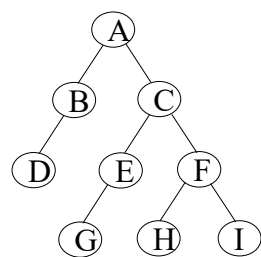
```
typedef struct BiThrNode
{
    ElemType data;
    struct BiTreeNode *Lchild, *Rchild;
    int Ltag, Rtag;
}BiThrNode;
```

如下图是二叉树及相应的各种线索树示例。

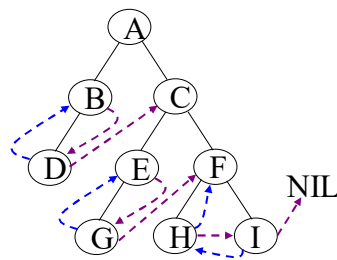
301

联航精英训练营

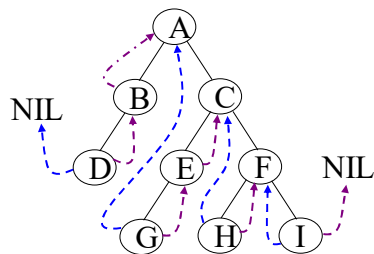
301



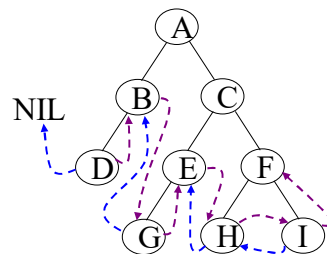
(a) 二叉树



(b) 先序线索树的逻辑形式  
结点序列: ABDCEGFHI



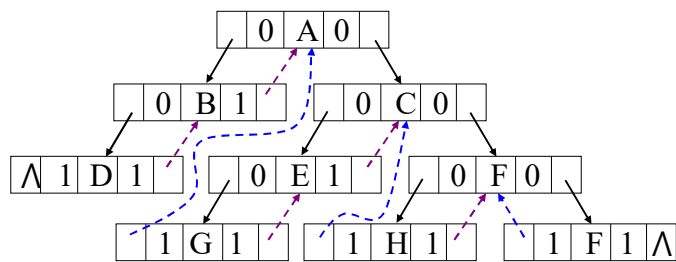
(c) 中序线索树的逻辑形式  
结点序列: DBAGECHFI



(d) 后序线索树的逻辑形式  
结点序列: DBGEHIFCA

302

302



(e) 中序线索树的链表结构

图 线索二叉树及其存储结构

**说明：**画线索二叉树时，**实线**表示指针，指向其左、右孩子；**虚线**表示线索，指向其直接前驱或直接后继。

在线索树上进行遍历，只要先找到序列中的第一个结点，然后就可以依次找结点的直接后继结点直到后继为空为止。



专业铸就品质 梦想成就未来

The Specialty Casts Quality, the Dream Gains Future.



## 第八章 查找

305



305

### 查找的概念

**查找表**(Search Table): 相同类型的数据元素(对象)组成的集合, 每个元素通常由若干数据项构成。

**关键字**(Key, 码): 数据元素中某个(或几个)数据项的值, 它可以标识一个数据元素。若关键字能**唯一**标识一个数据元素, 则关键字称为**主关键字**; 将能标识若干个数据元素的关键字称为**次关键字**。

**查找/检索**(Searching): 根据给定的K值, 在查找表中确定一个关键字等于给定值的记录或数据元素。

- ◆ 查找表中**存在**满足条件的记录: 查找成功; 结果: 所查到的记录信息或记录在查找表中的位置。
- ◆ 查找表中**不存在**满足条件的记录: 查找失败。

306



306

## 查找有两种基本形式：静态查找和动态查找

**静态查找(Static Search)**：在查找时只对数据元素进行查询或检索，查找表称为静态查找表。

**动态查找(Dynamic Search)**：在实施查找的同时，插入查找表中不存在的记录，或从查找表中删除已存在的某个记录，查找表称为动态查找表。

查找的对象是查找表，采用何种查找方法，首先取决于查找表的组织。查找表是记录的集合，而集合中的元素之间是一种完全松散的关系，因此，**查找表是一种非常灵活的数据结构，可以用多种方式来存储。**

根据存储结构的不同，查找方法可分为三大类：

- ① **顺序表和链表的查找**：将给定的K值与查找表中记录的关键字**逐个进行比较**，找到要查找的记录；
- ② **散列表的查找**：根据给定的K值**直接访问**查找表，从而找到要查找的记录；
- ③ **索引查找表的查找**：首先根据索引确定待查找记录所在的块，然后再从块中找到要查找的记录。

307



联航精英训练营

307

## 查找方法评价指标

查找过程中主要操作是关键字的比较，查找过程中关键字的**平均比较次数(平均查找长度ASL: Average Search Length)**作为衡量一个查找算法效率高低的**标准**。ASL定义为：

$$ASL = \sum_{i=1}^n P_i \times C_i \quad \text{n为查找表中记录个数} \quad \sum_{i=1}^n P_i = 1$$

其中：

$P_i$ ：查找第*i*个记录的概率，不失一般性，认为查找每个记录的概率相等，即 $P_1=P_2=\dots=P_n=1/n$ ；

$C_i$ ：查找第*i*个记录需要进行比较的次数。

一般地，认为记录的关键字是一些可以进行比较运算的类型，如整型、字符型、实型等

308



联航精英训练营

308

## 静态查找

309

309

## 静态查找

静态查找表的抽象数据类型定义如下：

ADT Static\_SearchTable{

    数据对象D：D是具有相同特性的数据元素的集合，  
        各个数据元素有唯一标识的关键字。

    数据关系R：数据元素同属于一个集合。

    基本操作P：

    ⋮

} ADT Static\_SearchTable 。

线性表是查找表最简单的一种组织方式，本节介绍几种主要的关于顺序存储结构的查找方法。

310

310

## 顺序查找(Sequential Search)

### 1 查找思想

从表的一端开始逐个将记录的关键字和给定K值进行比较，若某个记录的关键字和给定K值相等，查找成功；否则，若扫描完整个表，仍然没有找到相应的记录，则查找失败。顺序表的类型定义如下：

```
#define MAX_SIZE 100

typedef struct SSTable
{
    RecType elem[MAX_SIZE]; /* 顺序表 */
    int length; /* 实际元素个数 */
}SSTable;
```

311



311

```
int Seq_Search(SSTable ST, KeyType key)
{
    int p;
    ST.elem[0].key=key; /* 设置监视哨兵,失败返回0 */
    for (p=ST.length; !EQ(ST.elem[p].key, key); p--);
    return(p);
}
```

比较次数:

查找第n个元素: 1

.....

查找第i个元素: n-i+1

查找第1个元素: n

查找失败: n+1

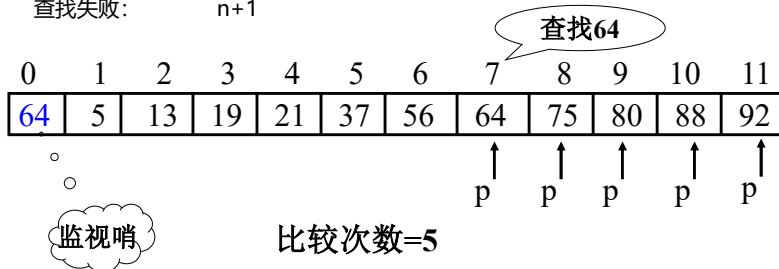


图 顺序查找示例

312

312

## 算法分析

不失一般性，设查找每个记录成功的概率相等，即 $P_i=1/n$ ；查找第 $i$ 个元素成功的比较次数 $C_i=n-i+1$ ；

◆ 查找成功时的平均查找长度ASL：

$$ASL = \sum_{i=1}^n P_i \times C_i = \frac{1}{n} \sum_{i=1}^n (n-i+1) = \frac{n+1}{2}$$

◆ 包含查找不成功时：查找失败的比较次数为 $n+1$ ，若成功与不成功的概率相等，对每个记录的查找概率为 $P_i=1/(2n)$ ，则平均查找长度ASL：

$$ASL = \sum_{i=1}^n P_i \times C_i = \frac{1}{2n} \sum_{i=1}^n (n-i+1) + \frac{n+1}{2} = 3(n+1)/4$$

313



联航精英训练营

313

## 折半查找(Binary Search)

折半查找又称为二分查找，是一种效率较高的查找方法。

前提条件：查找表中的所有记录是按关键字有序(升序或降序)。

查找过程中，先确定待查找记录在表中的范围，然后逐步缩小范围(每次将待查记录所在区间缩小一半)，直到找到或找不到记录为止。

314



联航精英训练营

314

## 1 查找思想

用Low、High和Mid表示待查找区间的下界、上界和中间位置指针，初值为Low=1，High=n。

- (1) 取中间位置Mid:  $Mid = \lfloor (Low + High) / 2 \rfloor$ ;
- (2) 比较中间位置记录的关键字与给定的K值:
  - ① 相等: 查找成功;
  - ② 大于: 待查记录在区间的前半段, 修改上界指针:  $High = Mid - 1$ , 转(1);
  - ③ 小于: 待查记录在区间的后半段, 修改下界指针:  $Low = Mid + 1$ , 转(1);

直到越界( $Low > High$ ), 查找失败。



## 算法实现

```
int Bin_Search(SSTable ST, KeyType key)
{ int Low=1, High=ST.length, Mid;
  while (Low<High)
  { Mid=(Low+High)/2;
    if (EQ(ST.elem[Mid].key, key))
      return(Mid);
    else if (LT(ST.elem[Mid].key, key))
      Low=Mid+1;
    else High=Mid-1;
  }
  return(0); /* 查找失败 */
}
```



## 算法分析

① 查找时每经过一次比较，查找范围就缩小一半，该过程可用一棵二叉树表示：

- ◆ 根结点就是第一次进行比较的中间位置的记录；
- ◆ 排在中间位置前面的作为左子树的结点；
- ◆ 排在中间位置后面的作为右子树的结点；

对各子树来说都是相同的。这样所得到的二叉树称为判定树(Decision Tree)。

② 将二叉判定树的第 $\lfloor \log_2 n \rfloor + 1$ 层上的结点补齐就成为一棵满二叉树，深度不变， $h = \lfloor \log_2(n+1) \rfloor$ 。

## 算法分析

③ 由满二叉树性质知，第 $i$ 层上的结点数为 $2^{i-1}$  ( $i \leq h$ )，设表中每个记录的查找概率相等，即 $P_i = 1/n$ ，查找成功时的平均查找长度ASL：

$$ASL = \sum_{i=1}^n P_i \times C_i = \frac{1}{n} \sum_{j=1}^h j \times 2^{j-1} = \frac{n+1}{n} \log_2(n+1) - 1$$

当 $n$ 很大 ( $n > 50$ ) 时， $ASL \approx \log_2(n+1) - 1$ 。

## 分块查找

**分块查找**(Blocking Search)又称**索引顺序查找**，是前面两种查找方法的综合。

### 1 查找表的组织

- ① 将查找表分成几块。**块间有序**，即第 $i+1$ 块的所有记录关键字均大于(或小于)第 $i$ 块记录关键字；**块内无序**。
- ② 在查找表的基础上附加一个索引表，索引表是按关键字有序的，索引表中记录的构成是：

最大关键字
起始指针

319

319

## 分块查找

### 2 查找思想

先确定待查记录所在块，再在块内查找(顺序查找)。

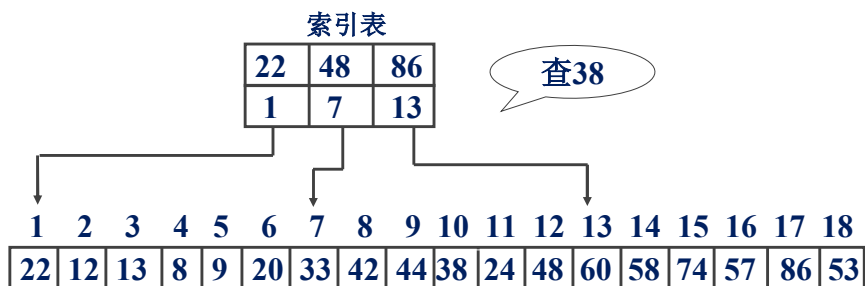


图 分块查找示例

320

320



## 算法实现

```
typedef struct IndexType
{ keyType maxkey; /* 块中最大的关键字 */
  int startpos; /* 块的起始位置指针 */
}Index;

/* 在分块索引表中查找关键字为key的记录，表长为n，块数为b */
int Block_search(RecType ST[], Index ind[], KeyType key, int n, int b)
{
    int i=0, j, k;
    while ((i<b)&&LT(ind[i].maxkey, key)) i++ ;
    if (i>b) { printf("\nNot found"); return(0); }

    j=ind[i].startpos;
    while ((j<n)&&LQ(ST[j].key, ind[i].maxkey) )
    { if ( EQ(ST[j].key, key) ) break ;
      j++ ; } /* 在块内查找 */

    if (j>n||!EQ(ST[j].key, key) )
    { j=0; printf("\nNot found"); }
    return(j);
}
```

321



联航精英训练营

321

## 算法分析

设表长为n个记录，均分为b块，每块记录数为s，则 $b=\lceil n/s \rceil$ 。设记录的查找概率相等，每块的查找概率为1/b，块中记录的查找概率为1/s，则平均查找长度ASL：

$$ASL=L_b+L_w=\sum_{j=1}^b j + \frac{1}{s} \sum_{i=1}^s i = \frac{b+1}{2} + \frac{s+1}{2}$$

**Fibonacci查找方法**是根据Fibonacci数列的特点对查找表进行分割。  
Fibonacci数列的定义是：

$$F(0)=0, F(1)=1, F(j)=F(j-1)+F(j-2)。$$

322



联航精英训练营

322

## 查找方法比较

	顺序查找	折半查找	分块查找
ASL	最大	最小	两者之间
表结构	有序表、无序表	有序表	分块有序表
存储结构	顺序存储结构 线性链表	顺序存储结构	顺序存储结构 线性链表

323



323

## 动态查找

324



324

## 动态查找

- 当查找表以线性表的形式组织时，若对查找表进行插入、删除或排序操作，就必须移动大量的记录，当记录数很多时，这种移动的代价很大。
- 利用树的形式组织查找表，可以对查找表进行动态高效的查找。

325

325

## 二叉排序树 (BST)

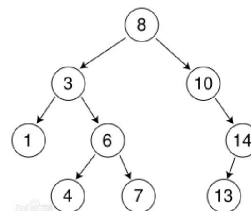
二叉排序树 (Binary Sort Tree)，又称二叉查找树 (Binary Search Tree)，亦称二叉搜索树。

二叉排序树或者是一棵空树，或者是具有下列性质的二叉树：

- (1) 若左子树不空，则左子树上所有结点的值 (关键字) **均小于** 它的根结点的值；
- (2) 若右子树不空，则右子树上所有结点的值 (关键字) **均大于** 它的根结点的值；
- (3) 左、右子树也分别为二叉排序树；

结论：若按中序遍历一棵二叉排序树，所得到的结点序列是一个递增序列。

BST仍然可以用二叉链表来存储，如右图所示。



326

326

## 平衡二叉树的定义

平衡二叉树或者是空树，或者是满足下列性质的二叉树。

- (1): 左子树和右子树深度之差的绝对值不大于1;
- (2): 左子树和右子树也都是平衡二叉树。

**平衡因子(Balance Factor)**：二叉树上结点的左子树的深度减去其右子树深度称为该结点的平衡因子。

因此，平衡二叉树上每个结点的平衡因子只可能是-1、0和1，否则，只要有一个结点的平衡因子的绝对值大于1，该二叉树就不是平衡二叉树。

如果一棵二叉树既是二叉排序树又是平衡二叉树，称为**平衡二叉排序树**(Balanced Binary Sort Tree)。

327



327

结点类型定义如下：

```
typedef struct BNode
{ KeyType key; /* 关键字域 */
  int Bfactor; /* 平衡因子域 */
  ... /* 其它数据域 */
  struct BNode *Lchild, *Rchild;
}BSTNode;
```

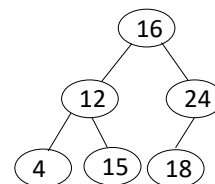


图 平衡二叉树

在平衡二叉排序树上执行查找的过程与二叉排序树上的查找过程完全一样，则在AVL树上执行查找时，和给定的K值比较的次数不超过树的深度。

设深度为h的平衡二叉排序树所具有的最少结点数为 $N_h$ ，则由平衡二叉排序树的性质知：

328

328

$$N_0 = 0, N_1 = 1, N_2 = 2, \dots, N_h = N_{h-1} + N_{h-2}$$

该关系和Fibonacci数列相似。根据归纳法可证明，当 $h \geq 0$ 时， $N_h = F_{h+2} - 1, \dots$ 而

$$F_h \approx \frac{\phi^h}{\sqrt{5}} \quad \text{其中 } \phi = \frac{1 + \sqrt{5}}{2} \quad \text{则 } N_h \approx \frac{\phi^h}{\sqrt{5}} - 1$$

这样，含有 $n$ 个结点的平衡二叉排序树的最大深度为

$$h \approx \log_{\phi} (\sqrt{5} \times (n+1)) - 2$$

则在平衡二叉排序树上进行查找的**平均查找长度**和 $\log_2 n$ 是一个数量级的，平均时间复杂度为 $O(\log_2 n)$ 。

## 索引查找

索引技术是**组织大型数据库**的重要技术，索引结构的基本组成是**索引表**和**数据表**两部分，如图所示。

- ◆ **数据表**：存储实际的数据记录；
- ◆ **索引表**：存储记录的**关键字**和记录(**存储**)**地址**之间的对照表，每个元素称为一个**索引项**。

通过索引表可实现对数据表中记录的快速查找。**索引表的组织有线性结构和树形结构**两种。

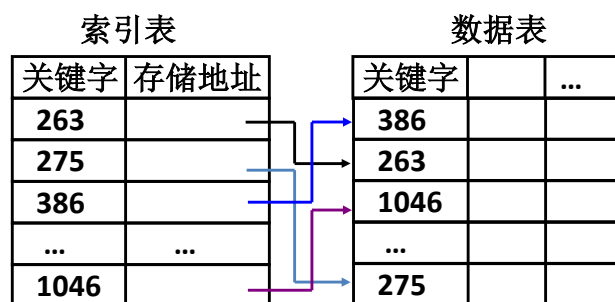


图 索引结构的基本形式

## 顺序索引表

是将索引项按顺序结构组织的线性索引表，而表中索引项一般是按关键字排序的，其特点是：

### 优点：

- ◆ 可以用折半查找方法快速找到关键字，进而找到数据记录的物理地址，实现数据记录的快速查找；
- ◆ 提供对变长数据记录的便捷访问；
- ◆ 插入或删除数据记录时不需要移动记录，但需要对索引表进行维护。

### 缺点：

- ◆ 索引表中索引项的数目与数据表中记录数相同，当索引表很大时，检索记录需多次访问外存；
- ◆ 对索引表的维护代价较高，涉及到大量索引项的移动，不适合于插入和删除操作。

## 树形索引表

平衡二叉排序树便于动态查找，因此用平衡二叉排序树来组织索引表是一种可行的选择。当用于大型数据库时，所有数据及索引都存储在外存，因此，涉及到内、外存之间频繁的数据交换，这种交换速度的快慢成为制约动态查找的瓶颈。若以二叉树的结点作为内、外存之间数据交换单位，则查找给定关键字时对磁盘平均进行 $\log_2 n$ 次访问是不能容忍的，因此，必须选择一种能尽可能降低磁盘I/O次数的索引组织方式。树结点的大小尽可能地接近页的大小。

R.Bayer和E.Mc Creight在1972年提出了一种多路平衡查找树，称为B\_树(其变型体是B+树)。

## B\_树

**B\_树**主要用于文件系统中，在B\_树中，每个结点的大小为一个磁盘页，结点中所包含的关键字及其孩子的数目取决于页的大小。一棵度为m的**B\_树**称为**m阶B\_树**，其定义是：

一棵m阶B\_树，或者是空树，或者是满足以下性质的m叉树：

- (1) 根结点或者是叶子，或者至少有两棵子树，至多有m棵子树；
- (2) 除根结点外，所有非终端结点至少有 $\lceil m/2 \rceil$ 棵子树，至多有m棵子树；
- (3) 所有叶子结点都在树的同一层上；
- (4) 每个结点应包含如下信息：

$(n, A_0, K_1, A_1, K_2, A_2, \dots, K_n, A_n)$

其中 $K_i (1 \leq i \leq n)$ 是**关键字**，且 $K_i < K_{i+1} (1 \leq i \leq n-1)$ ； $A_i (i=0, 1, \dots, n)$ 为**指向孩子结点的指针**，且 $A_{i-1}$ 所指向的子树中所有结点的关键字都小于 $K_i$ ， $A_i$ 所指向的子树中所有结点的关键字都大于 $K_i$ ；n是结点中关键字的个数，且 $\lfloor m/2 \rfloor - 1 \leq n \leq m-1$ ， $n+1$ 为子树的棵数。

333

333

当然，在实际应用中每个结点中还应包含n个指向每个关键字的记录指针，如图是一棵包含13个关键字的4阶B\_树。

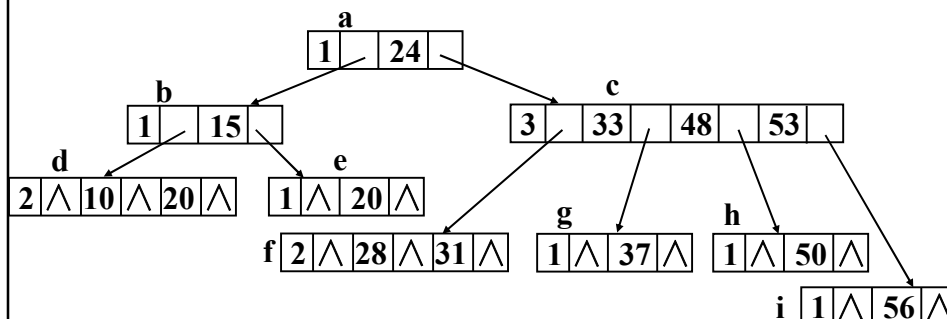


图 一棵包含13个关键字的4阶B\_树

334

334

## B+树

在实际的文件系统中，基本上不使用B\_树，而是使用B\_树的一种变体，称为m阶**B+树**。它与B\_树的主要不同是**叶子结点中存储记录**。在**B+树**中，所有的非叶子结点可以看成是索引，而其中的关键字是作为“分界关键字”，用来界定某一关键字的记录所在的子树。一棵m阶B+树与m阶B\_树的主要差异是：

- (1) 若一个结点有n棵子树，则必含有n个关键字；
- (2) 所有叶子结点中包含了全部记录的关键字信息以及这些关键字记录的指针，而且叶子结点按关键字的大小从小到大顺序链接；
- (3) 所有的非叶子结点可以看成是索引的部分，结点中只含有其子树的根结点中的最大(或最小)关键字。

335

335

## 一棵3阶B+树

如图是一棵3阶B+树。

由于B+树的叶子结点和非叶子结点结构上的显著区别，因此需要一个标志域加以区分，结点结构定义如下：

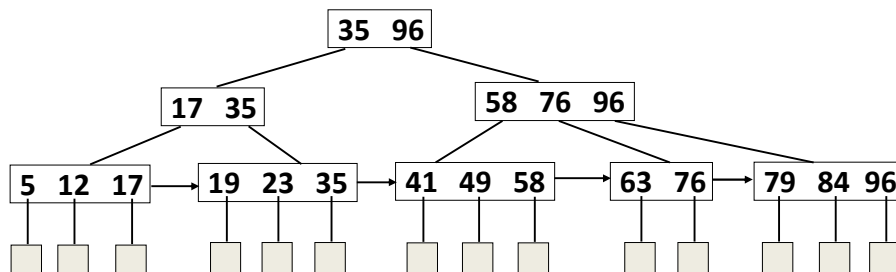


图 一棵3阶B+树

336

336



## 哈希查找

337

337

## 哈希(散列)查找

**基本思想：**在记录的存储地址和它的关键字之间建立一个确定的对应关系；这样，不经过比较，一次存取就能得到所查元素的查找方法。

例 30个地区的各民族人口统计表

编号	省、市(区)	总人口	汉族	回族	.....
1	北京				
2	上海				
.....	.....				

以编号作关键字，  
构造哈希函数： $H(\text{key})=\text{key}$   
 $H(1)=1$ ， $H(2)=2$

以地区别作关键字，取地区  
名称第一个拼音字母的序号  
作哈希函数： $H(\text{Beijing})=2$   
 $H(\text{Shanghai})=19$   $H(\text{Shenyang})=19$

338

338

## 哈希表

**哈希表**又称**散列表**。

**哈希表存储的基本思想是**：以数据表中的每个记录的关键字  $k$  为自变量，通过一种函数  $H(k)$  计算出函数值。把这个值解释为一块连续存储空间（即数组空间）的单元地址（即下标），将该记录存储到这个单元中。在此称该函数  $H$  为**哈希函数** 或**散列函数**。按这种方法建立的表称为**哈希表**或**散列表**。

例如，要将关键字值序列（3，15，22，24），存储到编号为0到4的表长为5的哈希表中。

计算存储地址的哈希函数可取除5的取余数算法  $H(k) = k \% 5$ 。则构造好的哈希表如图所示。

0	1	2	3	4
15		22	3	24

339



339

## 基本概念

**哈希函数**：在记录的关键字与记录的存储地址之间建立的一种对应关系叫哈希函数。

哈希函数是一种映象，是从关键字空间到存储地址空间的一种映象。可写成： $addr(a_i) = H(k_i)$ ，其中  $i$  是表中一个元素， $addr(a_i)$  是  $a_i$  的地址， $k_i$  是  $a_i$  的关键字。

**哈希表**：应用哈希函数，由记录的关键字确定记录在表中的地址，并将记录放入此地址，这样构成的表叫**哈希表**。

**哈希查找(又叫散列查找)**：利用哈希函数进行查找的过程叫**哈希查找**。

340



340

## 哈希表

**冲突**：对于不同的关键字 $k_i$ 、 $k_j$ ，若 $k_i \neq k_j$ ，但 $H(k_i) = H(k_j)$ 的现象叫冲突(collision)。

**同义词**：具有相同函数值的两个不同的关键字，称为该哈希函数的同义词。

哈希函数通常是一种压缩映象，所以冲突不可避免，只能尽量减少；当冲突发生时，应该有处理冲突的方法。

### 设计一个散列表应包括：

- ① 散列表的空间范围，即确定散列函数的值域；
- ② 构造合适的散列函数，使得对于所有可能的元素(记录的关键字)，函数值均在散列表的地址空间范围内，且出现冲突的可能尽量小；
- ③ 处理冲突的方法。即当冲突出现时如何解决。

341



联航精英训练营

341

## 哈希函数的构造

哈希函数是一种映象，其设定很灵活，只要使**任何关键字的哈希函数值都落在表长允许的范围之内**即可。

哈希函数“好坏”的主要评价因素有：

- ◆ 散列函数的**构造简单**；
- ◆ 能“**均匀**”地将散列表中的关键字映射到地址空间。所谓“**均匀**”(uniform)是指**发生冲突的可能性尽可能最少**。

342



联航精英训练营

342

## 哈希函数的构造方法

### 1 直接定址法

取关键字或关键字的某个线性函数作哈希地址，即 $H(\text{key})=\text{key}$  或 $H(\text{key})=a\cdot\text{key}+b$ ( $a, b$ 为常数)

**特点：**直接定址法所得地址集合与关键字集合大小相等，不会发生冲突，但实际中很少使用。

### 2 数字分析法

对关键字进行分析，取关键字的若干位或组合作为哈希地址。

适用于关键字位数比哈希地址位数大，且可能出现的关键字事先知道的情况。

343

343

## 哈希函数的构造方法

**例：**设有80个记录，关键字为8位十进制数，哈希地址为2位十进制数。

①②③④⑤⑥⑦⑧

8	1	3	4	6	5	3	2
8	1	3	7	2	2	4	2
8	1	3	8	7	4	2	2
8	1	3	0	1	3	6	7
8	1	3	2	2	8	1	7
8	1	3	3	8	9	6	7
8	1	3	6	8	5	3	7
8	1	4	1	9	3	5	5

分析：①只取8

②只取1

③只取3、4

⑧只取2、7、5

④⑤⑥⑦数字分布近乎随机

所以：取④⑤⑥⑦任意两位或两位与另两位的叠加作哈希地址

344

344

## 哈希函数的构造方法

### 3 平方取中法

将关键字平方后取中间几位作为哈希地址。

一个数平方后中间几位和数的每一位都有关，则由随机分布的关键字得到的散列地址也是随机的。散列函数所取的位数由散列表的长度决定。这种方法适于不知道全部关键字情况，是一种较为常用的方法。

### 4 折叠法

将关键字分割成位数相同的几部分(最后一部分可以不同)，然后取这几部分的叠加和作为哈希地址。

数位叠加有移位叠加和间界叠加两种。

345

345

## 哈希函数的构造方法

- ◆ 移位叠加：将分割后的几部分低位对齐相加。
- ◆ 间界叠加：从一端到另一端沿分割界来回折迭，然后对齐相加。

适于关键字位数很多，且每一位上数字分布大致均匀情况。

例：设关键字为0442205864，哈希地址位数为4。两种不同的地址计算方法如下：

$$\begin{array}{r} 5864 \\ 4220 \\ 04 \\ \hline 10088 \\ H(\text{key})=0088 \end{array}$$

移位叠加

$$\begin{array}{r} 5864 \\ 0224 \\ 04 \\ \hline 6092 \\ H(\text{key})=6092 \end{array}$$

间界叠加

346

346

## 哈希函数的构造方法

### 5 除留余数法

取关键字被某个不大于哈希表表长 $m$ 的数 $p$ 除后所得余数作哈希地址，即  
 $H(\text{key}) = \text{key} \text{ MOD } p \quad (p \leq m)$

是一种简单、常用的哈希函数构造方法。

利用这种方法的关键是 $p$ 的选取， $p$ 选的不好，容易产生同义词。 $p$ 的选取的分析：

- ◆ 选取 $p = 2^i (p \leq m)$ ：运算便于用移位来实现，但等于将关键字的高位忽略而仅留下低位二进制数。高位不同而低位相同的关键字是同义词。
- ◆ 选取 $p = q \times f$  ( $q, f$ 都是质因数,  $p \leq m$ )：则所有含有 $q$ 或 $f$ 因子的关键字的散列地址均是 $q$ 或 $f$ 的倍数。
- ◆ 选取 $p$ 为素数或 $p = q \times f$  ( $q, f$ 是质数且均大于20,  $p \leq m$ )：常用的选取方法，能减少冲突出现的可能性。

347



347

## 哈希函数的构造方法

### 6 随机数法

取关键字的随机函数值作哈希地址，即 $H(\text{key}) = \text{random}(\text{key})$   
当散列表中关键字长度不等时，该方法比较合适。

#### 选取哈希函数，考虑以下因素

- ◆ 计算哈希函数所需时间；
- ◆ 关键字的长度；
- ◆ 哈希表长度（哈希地址范围）；
- ◆ 关键字分布情况；
- ◆ 记录的查找频率。

348



348

## 哈希表冲突处理的方法

**冲突处理：**当出现冲突时，为冲突元素找到另一个存储位置。

### 1 开放定址法

**基本方法：**当冲突发生时，形成某个探测序列；按此序列逐个探测散列表中的其他地址，直到找到给定的关键字或一个空地址(开放的地址)为止，将发生冲突的记录放到该地址中。

散列地址的计算公式是：

$$H_i(\text{key}) = (H(\text{key}) + d_i) \text{ MOD } m, \quad i=1, 2, \dots, k(k \leq m-1)$$

其中：H(key)：哈希函数；m：散列表长度；

$d_i$ ：第*i*次探测时的增量序列；

$H_i(\text{key})$ ：经第*i*次探测后得到的散列地址。

349



349

## 哈希表冲突处理的方法

### (1) 线性探测法

将散列表T[0 ...m-1]看成循环向量。当发生冲突时，从初次发生冲突的位置依次向后探测其他的地址。

增量序列为： $d_i=1, 2, 3, \dots, m-1$

设初次发生冲突的地址是h，则依次探测T[h+1]，T[h+2]...，直到T[m-1]时又循环到表头，再次探测T[0]，T[1]...，直到T[h-1]。探测过程终止的情况是：

- ◆ **探测到的地址为空：**表中没有记录。若是查找则失败；若是插入则将记录写入到该地址；
- ◆ **探测到的地址有给定的关键字：**若是查找则成功；若是插入则失败；
- ◆ **直到T[h]：**仍未探测到空地址或给定的关键字，散列表满。

350



350

## 哈希表冲突处理的方法

例1：设散列表长为7，记录关键字组为：15, 14, 28, 26, 56, 23，散列函数： $H(\text{key}) = \text{key} \text{ MOD } 7$ ，冲突处理采用线性探测法。

解： $H(15) = 15 \text{ MOD } 7 = 1$

$H(14) = 14 \text{ MOD } 7 = 0$

$H(28) = 28 \text{ MOD } 7 = 0$  冲突  $H_1(28) = 1$  又冲突  $H_2(28) = 2$

$H(26) = 26 \text{ MOD } 7 = 5$

$H(56) = 56 \text{ MOD } 7 = 0$  冲突  $H_1(56) = 1$  又冲突  $H_2(56) = 2$  又冲突  $H_3(56) = 3$

$H(23) = 23 \text{ MOD } 7 = 2$  冲突  $H_1(23) = 3$  又冲突  $H_3(23) = 4$

0	1	2	3	4	5	6
14	15	28	56	23	26	

线性探测法的特点

- ♦ 优点：只要散列表未满，总能找到一个不冲突的散列地址；
- ♦ 缺点：每个产生冲突的记录被散列到离冲突最近的空地址上，从而又增加了更多的冲突机会(这种现象称为冲突的“聚集”)。

351



351

## 哈希表冲突处理的方法

### (2) 二次探测法

增量序列为： $d_i = 1^2, -1^2, 2^2, -2^2, 3^2, \dots, \pm k^2$  ( $k \leq \lfloor m/2 \rfloor$ )

上述例题若采用二次探测法进行冲突处理，则：

$H(15) = 15 \text{ MOD } 7 = 1$

$H(14) = 14 \text{ MOD } 7 = 0$

$H(28) = 28 \text{ MOD } 7 = 0$  冲突  $H_1(28) = 1$  又冲突  $H_2(28) = 4$

$H(26) = 26 \text{ MOD } 7 = 5$

$H(56) = 56 \text{ MOD } 7 = 0$  冲突  $H_1(56) = 1$  又冲突  $H_2(56) = 4$  又冲突  $H_3(56) = 9$  又冲突  $H_4(56) = 16$

$H(23) = 23 \text{ MOD } 7 = 2$  冲突  $H_1(23) = 3$

二次探测法的特点

- ♦ 优点：探测序列跳跃式地散列到整个表中，不易产生冲突的“聚集”现象；
- ♦ 缺点：不能保证探测到散列表的所有地址。

0	1	2	3	4	5	6
14	15	56	23	28	26	

352



352



## 哈希表冲突处理的方法

### (3) 伪随机探测法

增量序列使用一个伪随机函数来产生一个落在闭区间 $[1, m-1]$ 的随机序列。

例2：表长为11的哈希表中已填有关键字为17, 60, 29的记录，散列函数为 $H(\text{key}) = \text{key} \text{ MOD } 11$ 。现有第4个记录，其关键字为38，按三种处理冲突的方法，将它填入表中。

(1)  $H(38) = 38 \text{ MOD } 11 = 5$  冲突

$H_1 = (5+1) \text{ MOD } 11 = 6$  冲突

$H_2 = (5+2) \text{ MOD } 11 = 7$  冲突

$H_3 = (5+3) \text{ MOD } 11 = 8$  不冲突

(2)  $H(38) = 38 \text{ MOD } 11 = 5$  冲突

$H_1 = (5+1^2) \text{ MOD } 11 = 6$  冲突

$H_2 = (5-1^2) \text{ MOD } 11 = 4$  不冲突

(3)  $H(38) = 38 \text{ MOD } 11 = 5$  冲突

设伪随机数序列为9，则 $H_1 = (5+9) \text{ MOD } 11 = 3$  不冲突

0	1	2	3	4	5	6	7	8	9	10
			38	38	60	17	29	38		

353



联航精英训练营

353

## 哈希表冲突处理的方法

### 2 再哈希法

构造若干个哈希函数，当发生冲突时，利用不同的哈希函数再计算下一个新哈希地址，直到不发生冲突为止。即： $H_i = RH_i(\text{key}) \quad i=1, 2, \dots, k$

$RH_i$ ：一组不同的哈希函数。第一次发生冲突时，用 $RH_1$ 计算，第二次发生冲突时，用 $RH_2$ 计算...依此类推知道得到某个 $H_i$ 不再冲突为止。

- ◆ 优点：不易产生冲突的“聚集”现象；
- ◆ 缺点：计算时间增加。

354



联航精英训练营

354

## 哈希表冲突处理的方法

### 3 链地址法

**方法：**将所有关键字为同义词(散列地址相同)的记录存储在一个单链表中，并用一维数组存放链表的头指针。

设散列表长为m，定义一个一维指针数组：

$\text{RecNode}^* \text{linkhash}[m]$ ，其中RecNode是结点类型，每个分量的初值为空。凡散列地址为k的记录都插入到以linkhash[k]为头指针的链表中，插入位置可以在表头或表尾或按关键字排序插入。

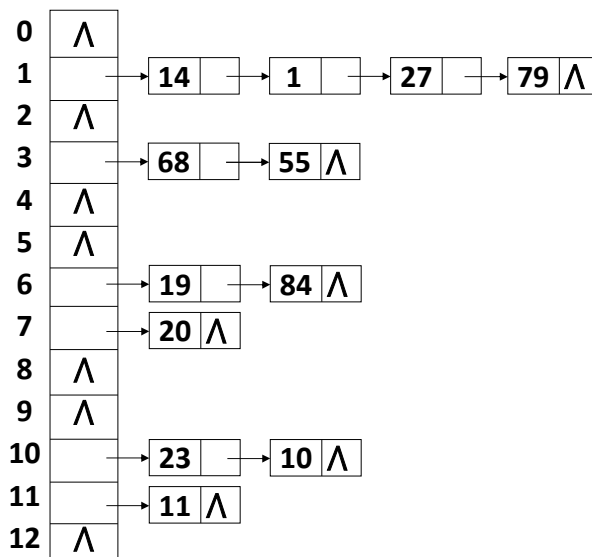
例：已知一组关键字(19, 14, 23, 1, 68, 20, 84, 27, 55, 11, 10, 79)，哈希函数为： $H(\text{key}) = \text{key} \text{ MOD } 13$ ，用链地址法处理冲突，如下图所示。

**优点：**不易产生冲突的“聚集”；删除记录也很简单。

355



355



用链地址法处理冲突的散列表

356

356

## 哈希表冲突处理的方法

### 4. 建立公共溢出区

**方法：**在基本散列表之外，另外设立一个溢出表保存与基本表中记录冲突的所有记录。

设散列表长为 $m$ ，设立基本散列表 $hashtable[m]$ ，每个分量保存一个记录；溢出表 $overtable[m]$ ，一旦某个记录的散列地址发生冲突，都填入溢出表中。

例：已知一组关键字(15, 4, 18, 7, 37, 47)，散列表长度为7，哈希函数为： $H(key)=key \text{ MOD } 7$ ，用建立公共溢出区法处理冲突。得到的基本表和溢出表如下：

Hashtable表：	散列地址	0	1	2	3	4	5	6
	关键字	7	15	37		4	47	
overtable表：	溢出地址	0	1	2	3	4	5	6
	关键字	18						

357

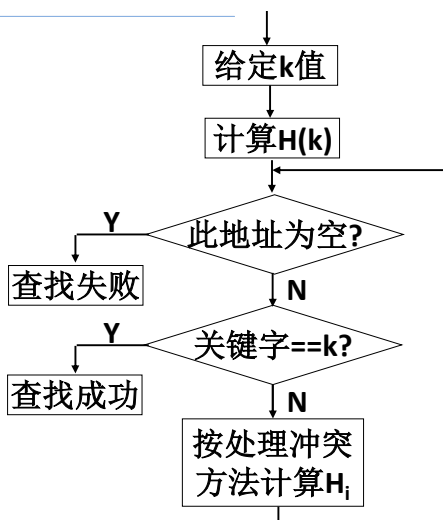
357

## 哈希查找过程及分析

### 1 哈希查找过程

哈希表的主要目的是用于快速查找，且插入和删除操作都要用到查找。由于散列表的特殊组织形式，其查找有特殊的方法。

设散列为 $HT[0 \dots m-1]$ ，散列函数为 $H(key)$ ，解决冲突的方法为 $R(x, i)$ ，则在散列表上查找定值为 $K$ 的记录的过程如图所示。



散列表的查找过程

358

358

## 哈希查找过程及分析

### 2 哈希查找分析

从哈希查找过程可见：尽管散列表在关键字与记录的存储地址之间建立了直接映像，但由于“冲突”，查找过程仍是一个给定值与关键字进行比较的过程，评价哈希查找效率仍要用ASL。

哈希查找时关键字与给定值比较的次数取决于：

- ◆ 哈希函数；
- ◆ 处理冲突的方法；
- ◆ 哈希表的填满因子 $\alpha$ 。填满因子 $\alpha$ 的定义是：

$$\alpha = \frac{\text{表中填入的记录数}}{\text{哈希表长度}}$$

359



联航精英训练营

359

### 各种散列函数所构造的散列表的ASL如下：

(1) 线性探测法的平均查找长度是：

$$S_{\text{nl成功}} \approx \frac{1}{2} \times (1 + \frac{1}{1-\alpha})$$

$$S_{\text{nl失败}} \approx \frac{1}{2} \times (1 + \frac{1}{(1-\alpha)^2})$$

(2) 二次探测、伪随机探测、再哈希法的平均查找长度是：

$$S_{\text{nl成功}} \approx -\frac{1}{\alpha} \times \ln(1-\alpha)$$

$$S_{\text{nl失败}} \approx \frac{1}{1-\alpha}$$

(3) 用链地址法解决冲突的平均查找长度是：

$$S_{\text{nl成功}} \approx 1 + \frac{\alpha}{2}$$

$$S_{\text{nl失败}} \approx \alpha + e^{-\alpha}$$

360



联航精英训练营

360

## Hash算法有什么特点

一个优秀的 hash 算法，将能实现：

**正向快速：**给定明文和 hash 算法，在有限时间和有限资源内能计算出 hash 值。

**逆向困难：**给定（若干） hash 值，在有限时间内很难（基本不可能）逆推出明文。

**输入敏感：**原始输入信息修改一点信息，产生的 hash 值看起来应该都有很大不同。

**冲突避免：**很难找到两段内容不同的明文，使得它们的 hash 值一致（发生冲突）。即对于任意两个不同的数据块，其hash值相同的可能性极小；对于一个给定的数据块，找到和它hash值相同的数据块极为困难。

- 但在不同的使用场景中，如数据结构和安全领域里，其中对某一些特点会有所侧重。

## Hash在密码学中的应用

- 在密码学中，hash算法的作用主要是用于**消息摘要**和**签名**，换句话说，它主要用于对整个消息的完整性进行校验。
- 举个例子，我们登陆网站的时候都需要输入密码，那么网站如果明文保存这个密码，那么黑客就很容易窃取大家的密码来登陆，特别不安全。
- 那么就想到了一个方法，使用hash算法生成一个密码的签名，后台只保存这个签名值。由于hash算法是不可逆的，那么黑客即便得到这个签名，也丝毫没有用处；而如果你在网站登陆界面上输入你的密码，那么知乎后台就会重新计算一下这个hash值，与网站中储存的原hash值进行比对，如果相同，证明你拥有这个账户的密码，那么就会允许你登陆。
- 银行也是如此，银行是万万不敢保存用户密码的原文的，只会保存密码的hash值而已。

## Hash在密码学中的应用

- 在这些应用场景里，对于抗碰撞和抗篡改能力要求极高，对速度的要求在其次。
- 一个设计良好的hash算法，其抗碰撞能力是很高的。
- 以MD5为例，其输出长度为128位，设计预期碰撞概率为，这是一个极小极小的数字为 $1/2$ 的64次方——而即便是在MD5被王小云教授破解之后，其碰撞概率上限也高达 $1/2$ 的41次方。而对于两个相似的字符串，MD5加密结果如下：

MD5("version1") = "966634ebf2fc135707d6753692bf4b1e";

MD5("version2") = "2e0e95285f08a07dea17e7ee111b21c8";

## MD5

- **MD5消息摘要算法**（英语：MD5 Message-Digest Algorithm），一种被广泛使用的密码散列函数，可以产生出一个128位（16字节）的散列值（hash value），用于确保信息传输完整一致。MD5由美国密码学家罗纳德·李维斯特（Ronald Linn Rivest）设计，于1992年公开，用以取代MD4算法。这套算法的程序在 RFC 1321 中被加以规范。
- **将数据（如一段文字）运算变为另一固定长度值，是散列算法的基础原理。**
- 1996年后被证实存在弱点，可以被加以破解，对于需要高度安全性的数据，专家一般建议改用其他算法，如SHA-2。
- 2004年，证实MD5算法无法防止碰撞（collision），因此不适用于安全性认证，如SSL公开密钥认证或是数字签名等用途。

## MD5算法特点

MD5算法具有以下特点：

- 1、**压缩性**：任意长度的数据，算出的MD5值长度都是固定的。
- 2、**容易计算**：从原数据计算出MD5值很容易。
- 3、**抗修改性**：对原数据进行任何改动，哪怕只修改1个字节，所得到的MD5值都有很大区别。
- 4、**强抗碰撞**：已知原数据和其MD5值，想找到一个具有相同MD5值的数据（即伪造数据）是非常困难的

- 2009年，中国科学院的谢涛和冯登国仅用了 $2^{20.96}$ 的碰撞算法复杂度，破解了MD5的碰撞抵抗，该攻击在普通计算机上运行只需要数秒钟。

365



365

## MD5用途

- MD5由MD4、MD3、MD2改进而来，主要增强算法复杂度和不可逆性。
- 目前，MD5算法因其**普遍、稳定、快速**的特点，仍广泛应用于**普通数据的错误检查领域**。  
例如在一些BitTorrent下载中，**软件将通过计算MD5检验下载到的文件片段的完整性**。
- MD5已经广泛使用在为文件传输提供一定的可靠性方面。例如，服务器预先提供一个MD5校验和，用户下载完文件以后，用MD5算法计算下载文件的MD5校验和，然后通过检查这两个校验和是否一致，就能判断下载的文件是否出错。
- MD5亦有应用于部分网上GAME以保证GAME的公平性，原理是系统先在玩家下注前已生成该局的结果，将该结果的字符串配合一组随机字符串利用MD5加密，将该加密字符串于玩家下注前便显示给玩家，再在结果开出后将未加密的字符串显示给玩家，玩家便可利用MD5工具加密验证该字符串是否吻合。

366



366

## MD5散列

- 一般128位的MD5散列被表示为32位十六进制数字。以下是一个43位长的仅ASCII字母列的MD5散列：

```
MD5("The quick brown fox jumps over the lazy dog") = 9e107d9d372bb6826bd81d3542a419d6
```

即使在原文中作一个小变化（比如用c取代d）其散列也会发生巨大的变化：

```
MD5("The quick brown fox jumps over the lazy cog") = 1055d3e698d289f2af8663725127bd4b
```

367



367

## SHA家族

- **安全散列算法**（英语：Secure Hash Algorithm，缩写为SHA）是一个密码散列函数家族，是FIPS所认证的安全散列算法。能计算出一个数字消息所对应的，长度固定的字符串（又称消息摘要）的算法。且若输入的消息不同，它们对应到不同字符串的机率很高。
- SHA家族的算法，由美国国家安全局（NSA）所设计，并由美国国家标准与技术研究院（NIST）发布，是美国的政府标准，其分别是：
  - ◆ **SHA-0**：1993年发布，当时称做安全散列标准（Secure Hash Standard），发布之后很快就被NSA撤回，是SHA-1的前身。
  - ◆ **SHA-1**：1995年发布，SHA-1在许多安全协议中广为使用，包括TLS和SSL、PGP、SSH、S/MIME和IPsec，曾被视为是MD5（更早之前被广为使用的散列函数）的后继者。但SHA-1的安全性在2000年以后已经不被大多数的加密场景所接受。2017年荷兰密码学研究小组CWI和Google正式宣布攻破了SHA-1。
  - ◆ **SHA-2**：2001年发布，包括SHA-224、SHA-256、SHA-384、SHA-512、SHA-512/224、**SHA-512/256**。虽然至今尚未出现对SHA-2有效的攻击，它的算法跟SHA-1基本上仍然相似；因此有些人开始发展其他替代的散列算法。
  - ◆ **SHA-3**：2015年正式发布，SHA-3并不是要取代SHA-2，因为SHA-2目前并没有出现明显的弱点。由于对MD5出现成功的破解，以及对SHA-0和SHA-1出现理论上破解的方法，NIST感觉需要一个与之前算法不同的，可替换的加密散列算法，也就是现在的SHA-3。

368



368



专业铸就品质 梦想成就未来

The Specialty Casts Quality,the Dream Gains Future.

## 第九章 排序

## 排序 (Sorting)

在信息处理过程中，最基本的操作是**查找**。从查找来说，效率最高的是**折半查找**，折半查找的前提是所有的数据元素(记录)是按关键字有序的。需要将一个无序的数据文件转变为一个有序的数据文件。

将任一文件中的记录通过某种方法整理成为按(记录)关键字有序排列的处理过程称为**排序**。

**排序**是数据处理中一种最常用的操作。

**排序**是将一批(组)任意次序的记录重新排列成**按关键字有序**的记录序列的过程。

371



371

## 排序的基本概念

### (1) 排序(Sorting)

**排序**是将一批(组)任意次序的记录重新排列成**按关键字有序**的记录序列的过程，其定义为：

给定一组记录序列： $\{R_1, R_2, \dots, R_n\}$ ，其相应的关键字序列是 $\{K_1, K_2, \dots, K_n\}$ 。确定 $1, 2, \dots, n$ 的一个排列 $p_1, p_2, \dots, p_n$ ，使其相应的关键字满足如下**非递减(或非递增)**关系： $K_{p_1} \leq K_{p_2} \leq \dots \leq K_{p_n}$ 的序列 $\{K_{p_1}, K_{p_2}, \dots, K_{p_n}\}$ ，这种操作称为**排序**。

关键字 $K_i$ 可以是记录 $R_i$ 的主关键字，也可以是次关键字或若干数据项的组合。

- ◆  $K_i$ 是主关键字：排序后得到的结果是唯一的；
- ◆  $K_i$ 是次关键字：排序后得到的结果是不唯一的。

372



372

## 排序的基本概念

### (2) 排序的稳定性

若记录序列中有**两个或两个以上关键字相等**的记录： $K_i = K_j (i \neq j, i, j = 1, 2, \dots, n)$ ，且在排序前 $R_i$ 先于 $R_j (i < j)$ ，排序后的记录序列仍然是 $R_i$ 先于 $R_j$ ，称**排序方法是稳定的**，否则是不稳定的。

排序算法有许多，但就全面性能而言，还没有一种公认为最好的。每种算法都有其优点和缺点，分别适合不同的数据量和硬件配置。

评价排序算法的标准有：**执行时间**和**所需的辅助空间**，其次是**算法的稳定性**。

## 排序稳定性的意义

- 1、如果**只是简单的进行数字的排序**，那么稳定性将毫无意义。
- 2、如果排序的内容仅仅是一个复杂对象的**某一个数字属性**，那么稳定性依旧将毫无意义。
- 3、如果要排序的内容是一个复杂对象的多个数字属性，**但是其原本的初始顺序毫无意义**，那么稳定性依旧将毫无意义。
- 4、除非要排序的内容是一个复杂对象的多个数字属性，且其原本的初始顺序存在意义，那么**我们需要在二次排序的基础上保持原有排序的意义**，才需要使用到稳定性的算法，例如要排序的内容是一组原本按照价格高低排序的对象，如今需要按照销量高低排序，使用稳定性算法，可以使得想同销量的对象依旧保持着价格高低的排序展现，只有销量不同的才会重新排序。（当然，如果需求不需要保持初始的排序意义，那么使用稳定性算法依旧将毫无意义）。

## 排序的基本概念

若排序算法所需的辅助空间不依赖问题的规模 $n$ ，即空间复杂度是 $O(1)$ ，则称排序方法是**就地排序**，否则是**非就地排序**。

### (3) 排序的分类

待排序的记录数量不同，排序过程中涉及的存储器的不同，有不同的排序分类。

- ① **待排序的记录数不太多**：所有的记录都能存放在内存中进行排序，称为**内部排序**；
- ② **待排序的记录数太多**：所有的记录不可能存放在内存中，排序过程中必须在内、外存之间进行数据交换，这样的排序称为**外部排序**。

**外部排序**指的是大文件的排序，即待排序的记录存储在外存储器上，待排序的文件无法一次装入内存，需要在内存和外部存储器之间进行多次数据交换，以达到排序整个文件的目的。

外部排序最常用的算法是**多路归并排序**，即将原文件分解成多个能够一次性装入内存的部分分别把每一部分调入内存完成排序。然后，对已经排序的子文件进行归并排序。

375

## 排序的基本概念

### (4) 内部排序的基本操作

对内部排序地而言，其基本操作有两种：

- 1. **比较两个关键字的大小**；
- 2. **存储位置的移动：从一个位置移到另一个位置。**

第一种操作是必不可少的；而第二种操作却不是必须的，取决于记录的存储方式，具体情况是：

- ① **[顺序排序]记录存储在一组连续地址的存储空间**：记录之间的逻辑顺序关系是通过其物理存储位置的相邻来体现，**记录的移动是必不可少的**；
- ② **[链表排序]记录采用链式存储方式**：记录之间的逻辑顺序关系是通过结点中的指针来体现，排序过程**仅需修改结点的指针，而不需要移动记录**；
- ③ **[地址排序]记录存储在一组连续地址的存储空间**：构造另一个辅助表来保存各个记录的存放地址(指针)：排序过程**不需要移动记录**，而**仅需修改辅助表中的指针**，排序后视具体情况决定是否调整记录的存储位置。

①比较适合记录数较多的情况；而②、③则适合记录数较少的情况。

为讨论方便，假设待排序的记录是以①的情况存储，且设排序是按升序排列的；关键字是一些可直接用比较运算符进行比较的类型。

376



联航精英训练营

376

## 待排序的记录类型的定义

待排序的记录类型的定义如下：

```
#define MAX_SIZE 100
typedef int KeyType ;

typedef struct RecType
{ KeyType key ;          /* 关键字码 */
  infoType otherinfo ;  /* 其他域 */
}RecType ;

typedef struct Sqlist
{ RecType R[MAX_SIZE] ;
  int length ;
}Sqlist ;
```

377

377

## 排序

### 1. 什么是排序？

将一组杂乱无章的**数据**按一定的**规律**顺次排列起来。

存放在数据表中

按关键字排序

### 2. 排序的目的是什么？

——便于查找！

### 3. 排序算法的好坏如何衡量？

- ① **时间效率**——排序速度（即排序所花费的全部比较次数）
- ② **空间效率**——占内存辅助空间的大小
- ③ **稳定性**——若两个记录A和B的关键字值相等，但排序后A、B的先后次序保持不变，则称这种排序算法是稳定的。

378

378

## 排序

### 4. 什么叫内部排序?

——若待排序记录都在内存中，称为内部排序；

### 5. 待排序记录在内存中怎样存储和处理?

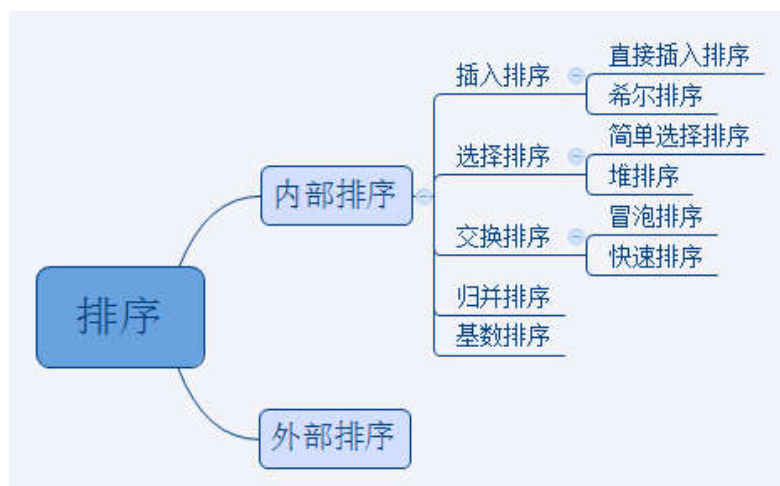
- ① 顺序排序——排序时直接移动记录；
- ② 链表排序——排序时只移动指针；
- ③ 地址排序——排序时先移动地址，最后再移动记录。

### 6. 内部排序的算法有哪些?

379

379

## 排序



380

380

## 插入排序

- 采用的是以“玩桥牌者”的方法为基础的。即在考察记录 $R_i$ 之前，设以前的所有记录 $R_1, R_2, \dots, R_{i-1}$ 已排好序，然后将 $R_i$ 插入到已排好序的诸记录的适当位置。
- 每步将一个待排序的对象，按其关键码大小，插入到前面**已经排好序**的一组对象的**适当位置**上，直到对象全部插入为止。

简言之，边插入边排序，保证子序列中随时都是排好序的。

- 最基本的插入排序是**直接插入排序**(Straight Insertion Sort)。

## 插入排序-直接插入排序

### 1 排序思想

将待排序的记录 $R_i$ ，插入到已排好序的记录表 $R_1, R_2, \dots, R_{i-1}$ 中，得到一个新的、记录数增加1的有序表。直到所有的记录都插入完为止。

设待排序的记录顺序存放在数组 $R[1\dots n]$ 中，在排序的某一时刻，将记录序列分成两部分：

- ◆  $R[1\dots i-1]$ ：已排好序的有序部分；
- ◆  $R[i\dots n]$ ：未排好序的无序部分。

显然，在刚开始排序时， $R[1]$ 是已经排好序的。

## 直接插入排序的例子

- 例：设有关键字序列为：7, 4, -2, 19, 13, 6, 直接插入排序的过程如下图所示：

初始记录的关键字： [7] 4 -2 19 13 6

第一趟排序： [4 7] -2 19 13 6

第二趟排序： [-2 4 7] 19 13 6

第三趟排序： [-2 4 7 19] 13 6

第四趟排序： [-2 4 7 13 19] 6

第五趟排序： [-2 4 6 7 13 19]

直接插入排序的过程

383



联航精英训练营

383

## 直接插入排序- 算法实现

```
void straight_insert_sort(Sqlist *L)
{
    int i, j;
    for (i=2; i<=L->length; i++)
    {
        L->R[0]=L->R[i]; j=i-1; /* 设置哨兵 */
        while( LT(L->R[0].key, L->R[j].key) )
        {
            L->R[j+1]=L->R[j];
            j--;
        } /* 查找插入位置 */
        L->R[j+1]=L->R[0]; /* 插入到相应位置 */
    }
}
```

384



联航精英训练营

384



## 直接插入排序-算法说明

算法中的R[0]开始时并不存放任何待排序的记录，引入的作用主要有两个：

- ① 不需要增加辅助空间：保存当前待插入的记录R[i]，R[i]会因为记录的后移而被占用；
- ② 保证查找插入位置的内循环总可以在超出循环边界之前找到一个等于当前记录的记录，起“哨兵监视”作用，避免在内循环中每次都要判断j是否越界。

385

385

## 算法分析

(1) **最好情况**：若待排序记录按关键字从小到大排列(正序)，算法中的内循环无须执行，则一趟排序时：关键字比较次数1次，记录移动次数2次(R[i]→R[0]，R[0]→R[j+1])。

则整个排序的关键字**比较次数**和**记录移动次数**分别是：

$$\text{比较次数: } \sum_{i=2}^n 1 = n-1 \quad \text{移动次数: } \sum_{i=2}^n 2 = 2(n-1)$$

386

386

**(2) 最坏情况：**若待排序记录按关键字从大到小排列(逆序)，则一趟排序时：算法中的内循环体执行*i*-1，关键字比较次数*i*次，记录移动次数*i*+1。

则就整个排序而言：

$$\begin{aligned} \text{比较次数: } \sum_{i=2}^n i &= \frac{(n-1)(n+1)}{2} \\ \text{移动次数: } \sum_{i=2}^n (i+1) &= \frac{(n-1)(n+4)}{2} \end{aligned}$$

一般地，认为待排序的记录可能出现的各种排列的概率相同，则取以上两种情况的平均值，作为排序的关键字比较次数和记录移动次数，约为  $n^2/4$ ，则**复杂度为 $O(n^2)$** 。

387

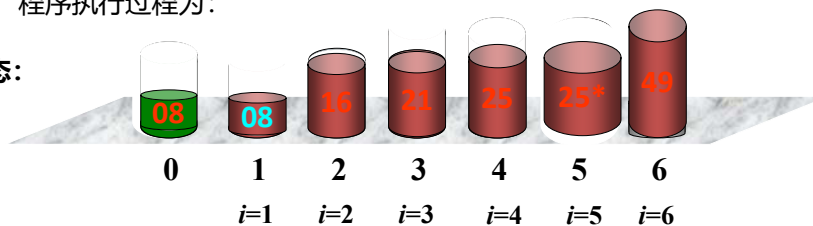
387

**例1：**关键字序列T= (21, 25, 49, 25\*, 16, 08)，

请写出直接插入排序的具体实现过程。 \*表示后一个25

**解：**假设该序列已存入一维数组r[7]中，将r[0]作为哨兵 (Temp)。则程序执行过程为：

**初态：**



**时间效率：**因为在最坏情况下，所有元素的比较次数总和为  $(0 + 1 + \dots + n - 1) \rightarrow O(n^2)$ 。其他情况下也要考虑移动元素的次数。故时间复杂度为  $O(n^2)$

**空间效率：**仅占用1个缓冲单元—— $O(1)$

**算法的稳定性：**因为25\*排序后仍然在25的后面——**稳定**

388

388

## 折半插入排序

**一个想得到的改进方法：**既然子表有序且为顺序存储结构，则插入时采用折半查找定可加速。

**优点：**比较次数大大减少，全部元素比较次数仅为 $O(n\log_2 n)$ 。

**时间效率：**虽然比较次数大大减少，可惜移动次数并未减少，所以排序效率仍为 $O(n^2)$ 。

**空间效率：**仍为 $O(1)$

**稳定性：**稳定

思考：折半插入排序还可以改进吗？  
能否减少移动次数？

### 2-路插入排序

这是对折半插入排序的一种改进，其目的是减少排序过程中的移动次数。

389



联航精英训练营

389

## 插入排序-折半插入排序

### 折半插入排序

当将待排序的记录 $R[i]$ 插入到已排好序的记录子表 $R[1...i-1]$ 中时，由于 $R_1, R_2, \dots, R_{i-1}$ 已排好序，则查找插入位置可以用“折半查找”实现，则直接插入排序就变成成为折半插入排序。

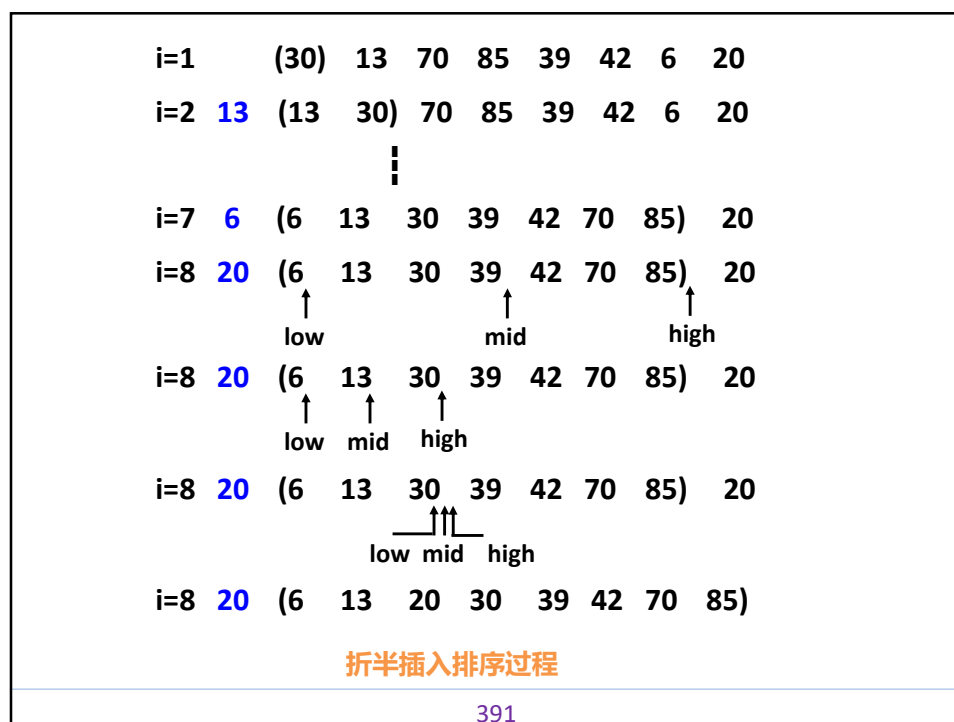
从时间上比较，折半插入排序仅仅减少了关键字的比较次数，却没有减少记录的移动次数，故时间复杂度仍然为 $O(n^2)$ 。

390



联航精英训练营

390



391


## 插入排序-希尔排序

**希尔排序**(Shell Sort, 又称**缩小增量法**)是一种**分组插入排序**方法。

### 1 排序思想

- ① 先取一个正整数 $d_1 (d_1 < n)$ 作为第一个增量, 将全部 $n$ 个记录分成 $d_1$ 组, 把所有相隔 $d_1$ 的记录放在一组中, 即对于每个 $k (k=1, 2, \dots, d_1)$ ,  $R[k], R[d_1+k], R[2d_1+k], \dots$ 分在同一组中, 在各组内进行直接插入排序。这样一次分组和排序过程称为一趟**希尔排序**;
- ② 取新的增量 $d_2 < d_1$ , 重复①的分组和排序操作; 直至所取的增量 $d_i=1$ 为止, 即所有记录放进一个组中排序为止。

392


**联航精英训练营**

392

## 排序示例

设有10个待排序的记录，关键字分别为9, 13, 8, 2, 5, 13, 7, 1, 15, 11，增量序列是5, 3, 1，希尔排序的过程如图所示。

初始关键字序列: 9   13   8   2   5   13   7   1   15   11

第一趟排序过程:



第一趟排序后: 9   7   1   2   5   13   13   8   15   11

第二趟排序后: 2   5   1   9   7   13   11   8   15   13

第三趟排序后: 1   2   5   7   8   9   11   13   13   15

希尔排序过程

393



联航精英训练营

393

## 算法实现

先给出一趟希尔排序的算法，类似直接插入排序。

/\* 对顺序表L进行一趟希尔排序，增量为d \*/

void shell\_pass(Sqlist \*L, int d)

{ int j, k;

for (j=d+1; j<=L->length; j++)

{ L->R[0]=L->R[j]; /\* 设置监视哨兵 \*/

k=j-d;

while (k>0&&LT(L->R[0].key, L->R[k].key) )

{ L->R[k+d]=L->R[k]; k=k-d; }

L->R[k+j]=L->R[0];

}

}

394



联航精英训练营

394

## 希尔排序

然后在根据增量数组dk进行希尔排序。

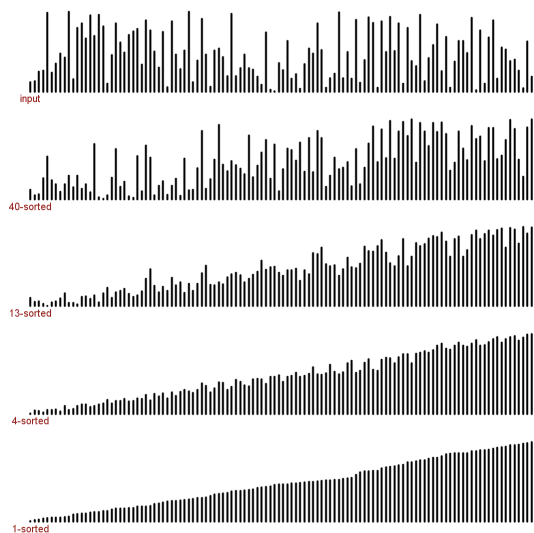
```
/* 按增量序列dk[0 ... t-1],对顺序表L进行希尔排序 */
void shell_sort(Sqlist *L, int dk[], int t)
{ int m;
  for (m=0; m<=t; m++)
    shell_pass(L, dk[m]);
}
```

希尔排序的分析比较复杂，涉及一些数学上的问题，其时间是所取的“增量”序列的函数。

395

395

## Visual trace of shell sort



Visual trace of shell sort

396

396

## 希尔排序特点

### 希尔排序特点:

子序列的构成不是简单的“逐段分割”，而是将相隔某个增量的记录组成一个子序列。

希尔排序可提高排序速度，原因是：

- ◆ 分组后 $n$ 值减小， $n^2$ 更小，而 $T(n) = O(n^2)$ ，所以 $T(n)$ 从总体上看是减小了；
- ◆ 关键字较小的记录跳跃式前移，在进行最后一趟增量为1的插入排序时，序列已基本有序。

### 增量序列取法

- ◆ 无除1以外的公因子；
- ◆ 最后一个增量值必须为1。

397



397

## 关于步长

- 希尔排序的关键并不是随便的分组后各自排序，而是将相隔某个“增量”的记录组成一个子序列，实现跳跃式的移动，使得排序的效率提高。
- 步长序列的不同，会导致最坏的时间复杂度情况的不同。
- 本文中，以 $N/2$ 为步长的最坏时间复杂度为 $N^2$ 。
- Donald Shell 最初建议步长选择为 $N/2$ 并且对步长取半直到步长达到1。虽然这样取可以比 $O(N^2)$ 类的算法（插入排序）更好，但这样仍然有减少平均时间和最差时间的余地。

步长序列	最坏情况下复杂度
$n/2^i$	$O(n^2)$
$2^k - 1$	$O(n^{3/2})$
$2^i 3^j$	$O(n \log^2 n)$

398



398

## 关于步长

- 已知的最好步长序列由**Marcin Ciura**设计 (1, 4, 10, 23, 57, 132, 301, 701, 1750, ...) 这项研究也表明 “比较在希尔排序中是最主要的操作，而不是交换。” 用这样步长序列的希尔排序比**插入排序**和**堆排序**都要快，甚至在小数组中比**快速排序**还快，但是在涉及大量数据时希尔排序还是比快速排序慢。
- 另一个在大数组中表现优异的步长序列是(**斐波那契数列**除去0和1将剩余的数以黄金分割比的两倍的幂进行运算得到的数列): (1, 9, 34, 182, 836, 4025, 19001, 90358, 428481, 2034035, 9651787, 45806244, 217378076, 1031612713, ...)

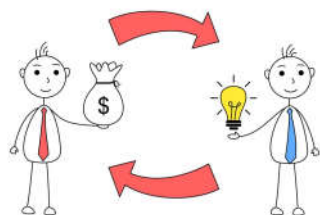
## 希尔排序小结

- **希尔排序**比**插入排序**和**选择排序**要快得多，并且数组越大，优势越大。希尔排序能够解决一些初级排序算法无能为力的问题。
- 有经验的程序员有时会选择希尔排序，因为对于**中等大小的数组**它的运行时间是可以接受的。
- **优点**：它的代码量很小，且不需要使用额外的内存空间。在后面的学习中我们会看到更加高效的算法，但除了对于很大的N，它们可能只会比希尔排序快两倍（可能还达不到），而且更复杂。
- **应用场景**：如果你需要解决一个排序问题而又没有系统排序函数可用（例如直接接触硬件或是运行于嵌入式系统中的代码），可以先用希尔排序，然后再考虑是否值得将它替换为更加复杂的排序算法。



## 交换排序

是一类基于交换的排序，系统地**交换反序的记录的对**，直到不再有这样一来的偶对为止。其中最基本的是**冒泡排序**(Bubble Sort)。



401

联航精英训练营

401

## 交换排序-冒泡排序

### 1 排序思想

依次比较相邻的两个记录的关键字，若两个记录是反序的(即前一个记录的关键字**大于**后一个记录的关键字)，则进行交换，直到没有反序的记录为止。

① 首先将 $L \rightarrow R[1]$ 与 $L \rightarrow R[2]$ 的关键字进行比较，若为反序( $L \rightarrow R[1]$ 的关键字大于 $L \rightarrow R[2]$ 的关键字)，则交换两个记录；然后比较 $L \rightarrow R[2]$ 与 $L \rightarrow R[3]$ 的关键字，依此类推，直到 $L \rightarrow R[n-1]$ 与 $L \rightarrow R[n]$ 的关键字比较后为止，称为一趟**冒泡排序**， $L \rightarrow R[n]$ 为关键字最大的记录。

② 然后进行第二趟**冒泡排序**，对前 $n-1$ 个记录进行同样的操作。

一般地，第 $i$ 趟冒泡排序是对 $L \rightarrow R[1 \dots n-i+1]$ 中的记录进行的，因此，若待排序的记录有 $n$ 个，则要经过 $n-1$ 趟冒泡排序才能使所有的记录有序。

402

联航精英训练营

402

## 冒泡排序的过程

初始关键字序列: 23 38 22 45 23 67 31 15 41

第一趟排序后: 23 22 38 23 45 31 15 41 67

第二趟排序后: 22 23 23 38 31 15 41 45 67

第三趟排序后: 22 23 23 31 15 38 41 45 67

第四趟排序后: 22 23 23 15 31 38 41 45 67

第五趟排序后: 22 23 15 23 31 38 41 45 67

第六趟排序后: 22 15 23 23 31 38 41 45 67

第七趟排序后: 15 22 23 23 31 38 41 45 67

图 冒泡排序过程

## 算法分析

### 时间复杂度

- ◆ 最好情况(正序): 比较次数:  $n-1$ ; 移动次数: 0;
- ◆ 最坏情况(逆序):

$$\text{比较次数: } \sum_{i=1}^{n-1} (n-i) = \frac{n(n-1)}{2}$$

$$\text{移动次数: } 3 \sum_{i=1}^{n-1} (n-i) = \frac{3n(n-1)}{2}$$

故时间复杂度:  $T(n) = O(n^2)$

空间复杂度:  $S(n) = O(1)$

稳定性: 稳定

## 交换排序-快速排序

**快速排序** (Quicksort) 是对冒泡排序的一种改进。

快速排序由C. A. R. Hoare在1962年提出。它的**基本思想**是：通过一趟排序，将待排序记录分割成独立的两部分，其中一部分记录的关键字均比另一部分记录的关键字小，再分别对这两部分记录进行下一趟排序，以达到整个序列有序。

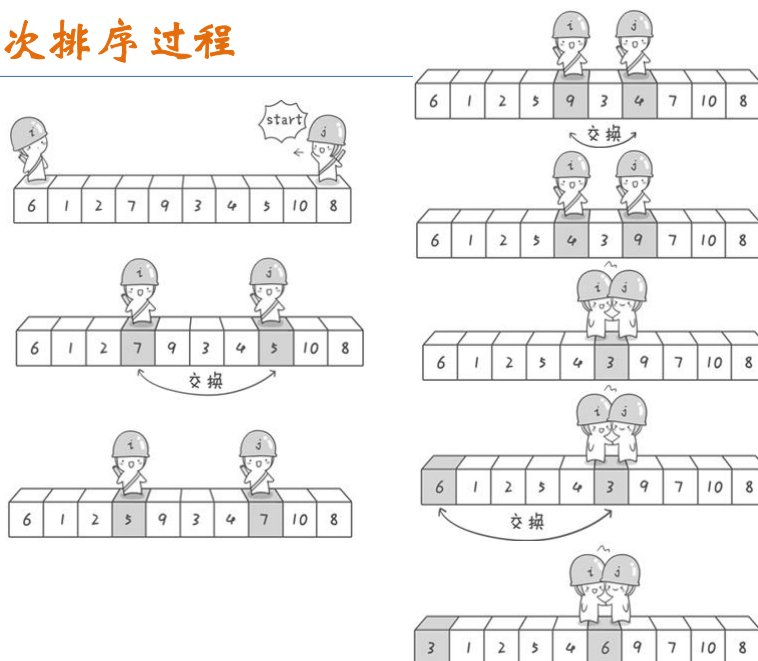
**一趟快速排序的算法是：**

- 1) 设置两个变量*i*、*j*，排序开始的时候： $i=0$ ， $j=N-1$ ；
- 2) 以第一个数组元素作为关键数据，赋值给**key**，即 $key=A[0]$ ；
- 3) 从*j*开始向前搜索，即由后开始向前搜索( $j--$ )，找到第一个小于**key**的值 $A[j]$ ，将 $A[j]$ 和 $A[i]$ 互换；
- 4) 从*i*开始向后搜索，即由前开始向后搜索( $i++$ )，找到第一个大于**key**的 $A[i]$ ，将 $A[i]$ 和 $A[j]$ 互换；
- 5) 重复第3、4步，直到 $i=j$ ；(3,4步中，没找到符合条件的值，即3中 $A[j]$ 不小于**key**，4中 $A[i]$ 不大于**key**的时候改变*j*、*i*的值，使得 $j=j-1$ ， $i=i+1$ ，直至找到为止。找到符合条件的值，进行交换的时候*i*，*j*指针位置不变。另外， $i=j$ 这一过程一定正好是*i*+或*j*-完成的时候，此时令循环结束)。

405

405

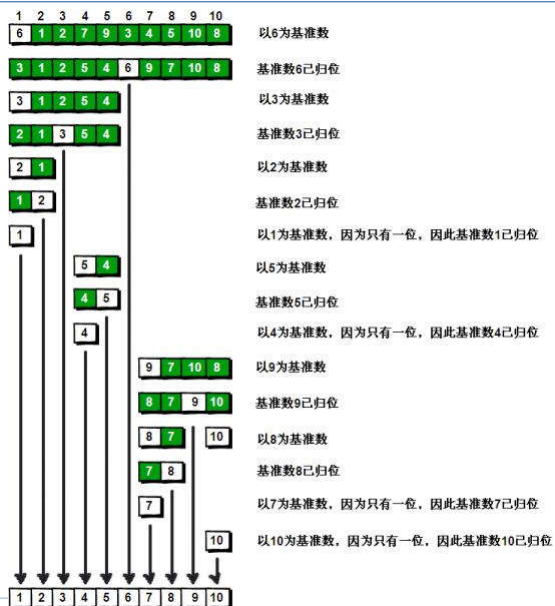
## 一次排序过程



406

406

## 快速排序过程



407

407

## 快速排序的时间复杂度

- 快速排序之所比较快，因为相比冒泡排序，每次交换是**跳跃式**的。每次排序的时候设置一个基准点，将小于等于基准点的数全部放到基准点的左边，将大于等于基准点的数全部放到基准点的右边。这样在每次交换的时候就不会像冒泡排序一样每次只能在相邻的数之间进行交换，**交换的距离就大的多了**。因此总的比较和交换次数就少了，速度自然就提高了。
- 其实快速排序是基于一种叫做“**二分**”的思想。

408

408

## 快速排序-算法分析

快速排序的主要时间是花费在划分上，对长度为k的记录序列进行划分时关键字的比较次数是k-1。设长度为n的记录序列进行排序的比较次数为C(n)，则 $C(n) = n - 1 + C(k) + C(n - k - 1)$ 。

**最好情况：**每次划分得到的子序列大致相等，则 $C(n) = n - 1 + C(k) + C(n - k - 1)$ 。

即 $C(n) \leq n \times \log_2 n + n \times C(1)$ ，C(1)看成常数因子，

即 $C(n) \leq O(n \times \log_2 n)$ ；

**最坏情况：**每次划分得到的子序列中有一个为空，另一个子序列的长度为n-1。

即每次划分所选择的基准是当前待排序序列中的最小(或最大)关键字

$$\text{比较次数: } \sum_{i=1}^{n-1} (n-i) = \frac{n(n-1)}{2} \quad \text{即 } C(n) = O(n^2)$$

快速排序的平均时间复杂度是： $T(n) = O(n \log_2 n)$

409



409

## 算法分析

从所需要的附加空间来看，快速排序算法是递归调用，系统内用堆栈保存递归参数，当每次划分比较均匀时，栈的最大深度为 $\lfloor \log_2 n \rfloor + 1$ 。

∴ 快速排序的空间复杂度是： $S(n) = O(\log_2 n)$

从排序的稳定性来看，快速排序是**不稳定的**。

410



410

## 选择排序

- **选择排序**(Selection Sort)的基本思想是：每次从当前待排序的记录中选取关键字最小的记录表，然后与待排序的记录序列中的第一个记录进行交换，直到整个记录序列有序为止。

411



411

## 选择排序-简单选择排序

**简单选择排序**(Simple Selection Sort，又称为**直接选择排序**)的基本操作是：通过 $n-i$ 次关键字间的比较，从 $n-i+1$ 个记录中选取关键字最小的记录，然后和第 $i$ 个记录进行交换， $i=1, 2, \dots, n-1$ 。

### 1 排序示例

例：设有关键字序列为：7, 4, -2, 19, 13, 6，直接选择排序的过程如下图所示。

初始记录的关键字： 7 4 -2 19 13 6

第一趟排序： -2 4 7 19 13 6

第二趟排序： -2 4 7 19 13 6

第三趟排序： -2 4 6 19 13 7

第四趟排序： -2 4 6 7 13 19

第五趟排序： -2 4 6 7 13 19

第六趟排序： -2 4 6 7 13 19

直接选择排序的过程

412



412

## 算法实现

```
void simple_selection_sort(Sqlist *L)
{ int m, n, k;
  for (m=1; m<L->length; m++)
  { k=m;
    for (n=m+1; n<=L->length; n++)
      if ( LT(L->R[n].key, L->R[k].key) ) k=n ;
    if (k!=m) /* 记录交换 */
      { L->R[0]=L->R[m]; L->R[m]=L->R[k];
        L->R[k]=L->R[0];
      }
  }
}
```

413



联航精英训练营

413

## 算法分析

整个算法是二重循环：外循环控制排序的趟数，对n个记录进行排序的趟数为n-1趟；内循环控制每一趟的排序。

进行第i趟排序时，关键字的比较次数为n-i，则：

$$\text{比较次数: } \sum_{i=1}^{n-1} (n-i) = \frac{n(n-1)}{2}$$

∴ 时间复杂度是： $T(n) = O(n^2)$

空间复杂度是： $S(n) = O(1)$

从排序的稳定性来看，直接选择排序是**不稳定**的。

414



联航精英训练营

414

## 树形选择排序

借助“淘汰赛”中的对垒就很容易理解树形选择排序的思想。

首先对 $n$ 个记录的关键字两两进行比较，选取 $\lceil n/2 \rceil$ 个较小者；然后这 $\lceil n/2 \rceil$ 个较小者两两进行比较，选取 $\lceil n/4 \rceil$ 个较小者... 如此重复，直到只剩1个关键字为止。

该过程可用一棵有 $n$ 个叶子结点的完全二叉树表示，如图所示。

每个枝结点的关键字都等于其左、右孩子结点中较小的关键字，**根结点的关键字就是最小的关键字。**

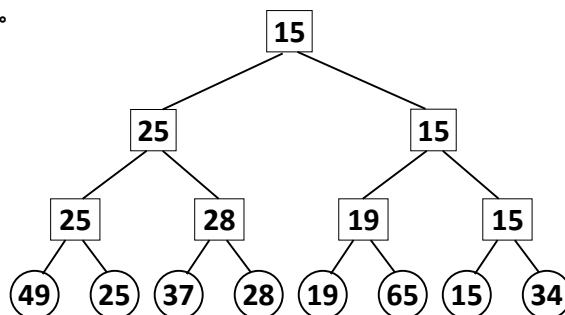
415

415

## 树形选择排序

输出最小关键字后，根据关系的可传递性，欲选取次小关键字，只需将叶子结点中的最小关键字改为“**最大值**”，然后重复上述步骤即可。

含有 $n$ 个叶子结点的完全二叉树的深度为 $\lceil \log_2 n \rceil + 1$ ，则总的时间复杂度为 $O(n \log_2 n)$ 。



图“淘汰赛”过程示意图

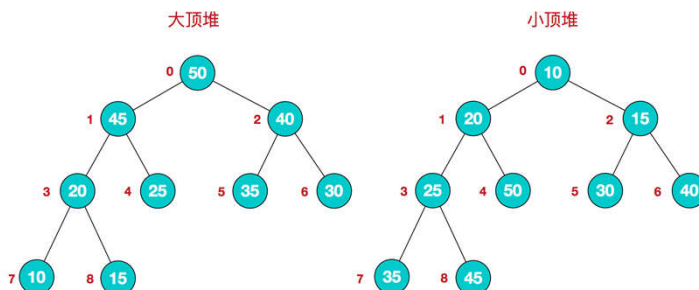
416

416



## 堆排序原理

堆是具有以下性质的完全二叉树：每个结点的值都大于或等于其左右孩子结点的值，称为大顶堆（大根堆）；或者每个结点的值都小于或等于其左右孩子结点的值，称为小顶堆（小根堆）。如下图：



同时，我们对堆中的结点按层进行编号，将这种逻辑结构映射到数组中就是下面这个样子

	0	1	2	3	4	5	6	7	8
arr	50	45	40	20	25	35	30	10	15

该数组从逻辑上讲就是一个堆结构，我们用简单的公式来描述一下堆的定义就是：

大顶堆：arr[i] >= arr[2i+1] && arr[i] >= arr[2i+2]

小顶堆：arr[i] <= arr[2i+1] && arr[i] <= arr[2i+2]

417

417

## 选择排序-堆排序

### 1 堆的定义

是n个元素的序列 $H = \{k_1, k_2, \dots, k_n\}$ ，满足：

$$\begin{cases} k_i \leq k_{2i} & \text{当 } 2i \leq n \text{ 时} \\ k_i \leq k_{2i+1} & \text{当 } 2i+1 \leq n \text{ 时} \end{cases} \quad \text{或} \quad \begin{cases} k_i \geq k_{2i} & \text{当 } 2i \leq n \text{ 时} \\ k_i \geq k_{2i+1} & \text{当 } 2i+1 \leq n \text{ 时} \end{cases}$$

其中：  $i=1, 2, \dots, \lfloor n/2 \rfloor$

由堆的定义知，堆是一棵以 $k_1$ 为根的完全二叉树。若对该二叉树的结点进行编号（从上到下，从左到右），得到的序列就是将二叉树的结点以顺序结构存放，堆的结构正好和该序列结构完全一致。

418

418

## 2 堆的性质

- ① 堆是一棵采用顺序存储结构的完全二叉树， $k_1$ 是根结点；
- ② 堆的根结点是关键字序列中的最小(或最大)值，分别称为小(或大)根堆；
- ③ 从根结点到每一叶子结点路径上的元素组成的序列都是按元素值(或关键字值)非递减(或非递增)的；
- ④ 堆中的任一子树也是堆。

利用堆顶记录的关键字值最小(或最大)的性质，从当前待排序的记录中**依次选取关键字最小(或最大)**的记录，就可以实现对数据记录的排序，这种排序方法称为**堆排序**。

## 3 堆排序思想

- ① 对一组待排序的记录，按堆的定义**建立堆**；
- ② 将**堆顶记录**和**最后一个记录**交换位置，则前 $n-1$ 个记录是无序的，而最后一个记录是有序的；
- ③ **堆顶记录**被交换后，前 $n-1$ 个记录不再是堆，需将前 $n-1$ 个待排序记录重新组织成为一个堆，然后将**堆顶记录**和**倒数第二个记录**交换位置，即将整个序列中次小关键字值的记录调整(排除)出无序区；
- ④ 重复上述步骤，直到全部记录排好序为止。

**结论：**排序过程中，若采用**小根堆**，排序后得到的是**非递增序列**；若采用**大根堆**，排序后得到的是**非递减序列**。

堆排序的关键

- 1. 如何由一个无序序列建成一个堆?
- 2. 如何在输出堆顶元素之后, 调整剩余元素, 使之成为一个新的堆?

4 堆的调整——筛选

(1) 堆的调整思想

输出堆顶元素之后, 以堆中最后一个元素替代之; 然后将根结点值与左、右子树的根结点值进行比较, 并与其中小者进行交换; 重复上述操作, 直到是叶子结点或其关键字值小于等于左、右子树的关键字的值, 将得到新的堆。称这个从堆顶至叶子的调整过程为“筛选”, 如图所示。

注意: 筛选过程中, 根结点的左、右子树都是堆, 因此, 筛选是从根结点到某个叶子结点的一次调整过程。

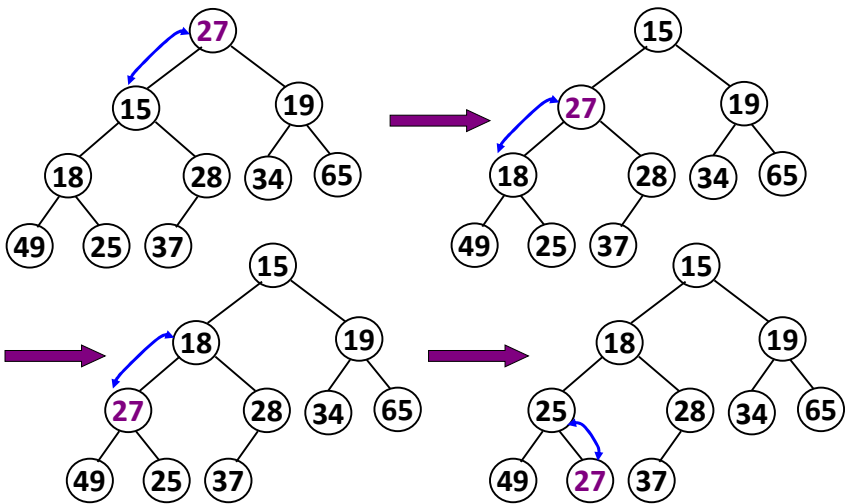


图 堆的筛选过程

## 算法分析

主要过程：初始建堆和重新调整成堆。设记录数为 $n$ ，所对应的完全二叉树深度为 $h$ 。

◆ **初始建堆**：每个非叶子结点都要从上到下做“筛选”。第 $i$ 层结点数 $\leq 2^{i-1}$ ，结点数下移的最大深度是 $h-i$ ，而每下移一层要比较2次，则比较次数 $C_1(n)$ 为：

$$C_1(n) \leq 2 \sum_{i=1}^{h-1} (2^{i-1} \times (h-i)) \leq 4(2^h - h - 1)$$

$$\because h = \lfloor \log_2 n \rfloor + 1, \therefore C_1(n) \leq 4(n - \log_2 n - 1)$$

423

423

## 算法分析

**筛选调整**：每次筛选要将根结点“下沉”到一个合适位置。第 $i$ 次筛选时：堆中元素个数为 $n-i+1$ ；堆的深度是 $\lfloor \log_2(n-i+1) \rfloor + 1$ ，则进行 $n-1$ 次“筛选”的比较次数 $C_2(n)$ 为：

$$C_2(n) \leq \sum_{i=1}^{n-1} (2 \times \log_2(n-i+1))$$

$$\therefore C_2(n) < 2n \log_2 n$$

$\therefore$  堆排序的比较次数的数量级为： $T(n) = O(n \log_2 n)$ ；而附加空间就是交换时所用的临时空间，故空间复杂度为： $S(n) = O(1)$ 。

稳定性：**不稳定**

424

424

## 归并排序

**归并(Merging)**：是指将两个或两个以上的有序序列合并成一个有序序列。若采用线性表(无论是那种存储结构)易于实现，其时间复杂度为 $O(m+n)$ 。

**归并思想实例**：两堆扑克牌，都已从小到大排好序，要将两堆合并为一堆且要求从小到大排序。

- ◆ 将两堆最上面的抽出(设为 $C_1$ ,  $C_2$ )比较大小，将小者置于一边作为新的一堆(不妨设 $C_1 < C_2$ )；再从第一堆中抽出一张继续与 $C_2$ 进行比较，将较小的放置在新堆的最下面；
- ◆ 重复上述过程，直到某一堆已抽完，然后将剩下一堆中的所有牌转移到新堆中。

425



425

## 归并排序-排序思想

- ① 初始时，将每个记录看成一个单独的有序序列，则 $n$ 个待排序记录就是 $n$ 个长度为1的有序子序列；
- ② 对所有有序子序列进行两两归并，得到 $\lceil n/2 \rceil$ 个长度为2或1的有序子序列——一趟归并；
- ③ 重复②，直到得到长度为 $n$ 的有序序列为止。

上述排序过程中，子序列总是两两归并，称为2-路归并排序。其核心是如何将相邻的两个子序列归并成一个子序列。设相邻的两个子序列分别为：

$\{R[k], R[k+1], \dots, R[m]\}$ 和 $\{R[m+1], R[m+2], \dots, R[h]\}$ ，将它们归并为一个有序的子序列：

$$\{DR[l], DR[l+1], \dots, DR[m], DR[m+1], \dots, DR[h]\}$$

426



426

## 归并排序-归并排序过程

例：设有9个待排序的记录，关键字分别为23, 38, 22, 45, 23, 67, 31, 15, 41，归并排序的过程如图所示。

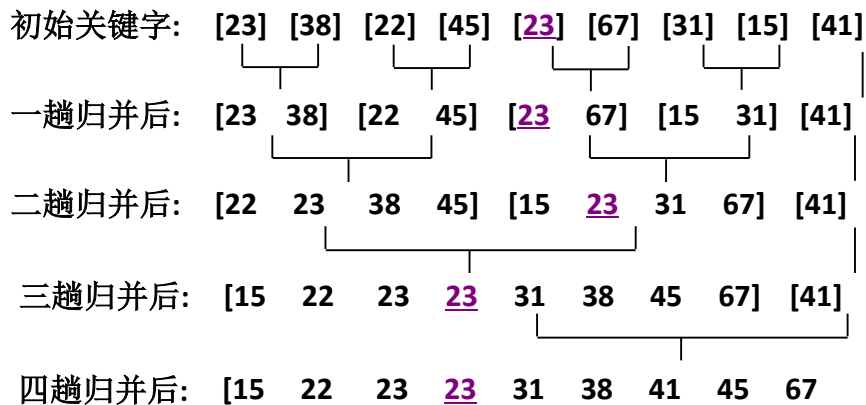


图 归并排序过程

427

427

## 归并的算法

```
void Merge(RecType R[], RecType DR[], int k, int m, int h)
{
    int p, q, n; p=n=k, q=m+1;
    while ((p<=m)&&(q<=h))
    {
        if (LQ(R[p].key, R[q].key)) /* 比较两个子序列 */
            DR[n++]=R[p++];
        else DR[n++]=R[q++];
    }
    while (p<=m) /* 将剩余子序列复制到结果序列中 */
        DR[n++]=R[p++];
    while (q<=h) DR[n++]=R[q++];
}
```

428

428

## 一趟归并排序

一趟归并排序都是从前到后，依次将相邻的两个有序子序列归并为一个，且除最后一个子序列外，其余每个子序列的长度都相同。设这些子序列的长度为 $d$ ，则一趟归并排序的过程是：

从 $j=1$ 开始，依次将相邻的两个有序子序列 $R[j\dots j+d-1]$ 和 $R[j+d\dots j+2d-1]$ 进行归并；每次归并两个子序列后， $j$ 后移动 $2d$ 个位置，即 $j=j+2d$ ；若剩下的元素不足两个子序列时，分以下两种情况处理：

- ① 剩下的元素个数 $>d$ ：再调用一次上述过程，将一个长度为 $d$ 的子序列和不足 $d$ 的子序列进行归并；
- ② 剩下的元素个数 $\leq d$ ：将剩下的元素依次复制到归并后的序列中。

## 一趟归并排序算法

```
void Merge_pass(RecType R[], RecType DR[], int d, int n)
{ int j=1;
  while ((j+2*d-1)<=n)
  { Merge(R, DR, j, j+d-1, j+2*d-1);
    j=j+2*d;
  } /* 子序列两两归并 */
  if (j+d-1<n) /* 剩余元素个数超过一个子序列长度d */
    Merge(R, DR, j, j+d-1, n);
  else Merge(R, DR, j, n, n); /* 剩余子序列复制 */
}
```

## 归并排序的算法

开始归并时，每个记录是长度为1的有序子序列，对这些有序子序列逐趟归并，每一趟归并后有序子序列的长度均扩大一倍；当有序子序列的长度与整个记录序列长度相等时，整个记录序列就成为有序序列。算法是：

```
void Merge_sort(Sqlist *L, RecType DR[])
{   int d=1;
    while(d<L->length)
    {   Merge_pass(L->R, DR, d, L->length);
        Merge_pass(DR, L->R, 2*d, L->length);
        d=4*d;
    }
}
```

431



联航精英训练营

431

## 归并排序-算法分析

具有 $n$ 个待排序记录的归并次数是 $\log_2 n$ ，而一趟归并的时间复杂度为 $O(n)$ ，则整个归并排序的时间复杂度无论是最好还是最坏情况均为 $O(n\log_2 n)$ 。在排序过程中，使用了辅助向量 $DR$ ，大小与待排序记录空间相同，则空间复杂度为 $O(n)$ 。

归并排序是稳定的。

432



联航精英训练营

432



## 基数排序

**基数排序**(Radix Sorting) 又称为**桶排序**或**数字排序**：按待排序记录的关键字的组成成分(或“位”)进行排序。

**基数排序**和前面的各种内部排序方法完全不同，**不需要进行关键字的比较和记录的移动**。借助于多关键字排序思想实现单逻辑关键字的排序。

433



433

## 多关键字排序

设有 $n$ 个记录 $\{R_1, R_2, \dots, R_n\}$ ，每个记录 $R_i$ 的关键字是由若干项(数据项)组成，即记录 $R_i$ 的关键字Key是若干项的集合： $\{Ki^1, Ki^2, \dots, Ki^d\} (d > 1)$ 。

记录 $\{R_1, R_2, \dots, R_n\}$ 有序的，指的是 $\forall i, j \in [1, n], i < j$ ，若记录的关键字满足：

$$\{Ki^1, Ki^2, \dots, Ki^d\} < \{Kj^1, Kj^2, \dots, Kj^d\},$$

即 $K_i^p \leq K_j^p (p = 1, 2, \dots, d)$

434



434

## 多关键字排序思想

先按第一个关键字 $K^1$ 进行排序，将记录序列分成若干个子序列，每个子序列有相同的 $K^1$ 值；然后分别对每个子序列按第二个关键字 $K^2$ 进行排序，每个子序列又被分成若干个更小的子序列；如此重复，直到按最后一个关键字 $K^d$ 进行排序。

最后，将所有的子序列依次联接成一个有序的记录序列，该方法称为**最高位优先(Most Significant Digit first)**。

另一种方法正好相反，排序的顺序是从最低位开始，称为**最低位优先(Least Significant Digit first)**。

435



435

## MSD与LSD

我们把扑克牌的排序看成由**花色**和**面值**两个数据项组成的主关键字排序。

要求如下：

花色顺序：梅花<方块<红心<黑桃

面值顺序：2<3<4<...<10<J<Q<K<A

那么，若要将一副扑克牌排成下列次序：

梅花2, ..., 梅花A, 方块2, ..., 方块A, 红心2, ..., 红心A, 黑桃2, ..., 黑桃A。

**有两种排序方法：**

<1> 先按花色分成四堆，把各堆收集起来；然后对每堆按面值由小到大排列，再按花色从小到大按堆收叠起来。----称为**"最高位优先"(MSD)**法。

<2> 先按面值由小到大排列成13堆，然后从小到大收集起来；再按花色不同分成四堆，最后顺序收集起来。----称为**"最低位优先"(LSD)**法。

436



436

## LSD

设有一个初始序列为: R {50, 123, 543, 187, 49, 30, 0, 2, 11, 100}。  
我们知道, 任何一个阿拉伯数, 它的各个位数上的基数都是以0~9来表示的。  
所以我们不妨把0~9视为10个桶。  
我们先根据序列的个位数的数字来进行分类, 将其分到指定的桶中。例如: R[0] = 50, 个位数上是0, 将这个数存入编号为0的桶中。

[0]	50	30	0	100
[1]	11			
[2]	2			
[3]	123	543		
[4]				
[5]				
[6]				
[7]	187			
[8]				
[9]	49			

分类后, 我们在从各个桶中, 将这些数按照从编号0到编号9的顺序依次将所有数取出来。  
这时, 得到的序列就是个位数上呈递增趋势的序列。  
按照个位数排序: {50, 30, 0, 100, 11, 2, 123, 543, 187, 49}。  
接下来, 可以对十位数、百位数也按照这种方法进行排序, 最后就能得到排序完成的序列。

437



437

## 算法分析-基数排序的性能

### 时间复杂度

通过上文可知, 假设在基数排序中,  $r$ 为基数,  $d$ 为位数。则基数排序的时间复杂度为 $O(d(n + rd))$ 。

我们可以看出, 基数排序的效率和初始序列是否有序没有关联。

### 空间复杂度

在基数排序过程中, 对于任何位数上的基数进行“装桶”操作时, 都需要 $n + rd$ 个临时空间, 空间复杂度为:  $O(n + rd)$ 。

### 算法稳定性

在基数排序过程中, 每次都是将当前位数上相同数值的元素统一“装桶”, 并不需要交换位置。所以基数排序是稳定的算法。

438



438

## 各种内部排序的比较

各种内部排序按所采用的基本思想(策略)可分为：**插入排序**、**交换排序**、**选择排序**、**归并排序**和**基数排序**，它们的基本策略分别是：

1. **插入排序**：依次将无序序列中的一个记录，按关键字值的大小插入到已排好序一个子序列的适当位置，直到所有的记录都插入为止。具体的方法有：**直接插入**、**折半法插入**、**表插入**、**2-路插入**和**shell排序**。
2. **交换排序**：对于待排序记录序列中的记录，两两比较记录的关键字，并对反序的两个记录进行交换，直到整个序列中没有反序的记录偶对为止。具体的方法有：**冒泡排序**、**快速排序**。

439

439

## 各种内部排序的比较

3. **选择排序**：不断地从待排序的记录序列中选取关键字最小的记录，放在已排好序的序列的最后，直到所有记录都被选取为止。具体的方法有：**简单选择排序**、**堆排序**。
4. **归并排序**：利用“归并”技术不断地对待排序记录序列中的有序子序列进行合并，直到合并为一个有序序列为止。
5. **基数排序**：按待排序记录的关键字的组成成分(“位”)从低到高(或从高到低)进行。每次是按记录关键字某一“位”的值将所有记录分配到相应的桶中，再按桶的编号依次将记录进行收集，最后得到一个有序序列。

各种内部排序方法的性能比较如下表。

440

440

## 主要内部排序方法的性能

方法	平均时间	最坏所需时间	附加空间	稳定性
直接插入	$O(n^2)$	$O(n^2)$	$O(1)$	稳定的
Shell排序	$O(n^{1.3})$	$O(n^2)$	$O(1)$	不稳定的
直接选择	$O(n^2)$	$O(n^2)$	$O(1)$	不稳定的
堆排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(1)$	不稳定的
冒泡排序	$O(n^2)$	$O(n^2)$	$O(1)$	稳定的
快速排序	$O(n\log_2 n)$	$O(n^2)$	$O(\log_2 n)$	不稳定的
归并排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n)$	稳定的
基数排序	$O(d(n+rd))$	$O(d(n+rd))$	$O(n+rd)$	稳定的

441

441

## 选取排序方法的主要考虑因素

- ◆ 待排序的记录数目 $n$ ;
- ◆ 每个记录的大小;
- ◆ 关键字的结构及其初始状态;
- ◆ 是否要求排序的稳定性;
- ◆ 语言工具的特性;
- ◆ 存储结构的初始条件和要求;
- ◆ 时间复杂度、空间复杂度和开发工作的复杂程度的平衡点等。

442

442



专业铸就品质 梦想成就未来

The Specialty Casts Quality,the Dream Gains Future.