# 数据结构

## 目录

1 来源：Didnelpsun/CS408: 考研计算机科学 408 笔记整理 (github.com)1

# 概述

## 基本概念

- 数据项：一个数据元素由若干个数据项组成。
- 数据元素：组成数据对象的基本单元。
- 数据对象：性质相同的数据元素的集合。

存储数据时要存储数据元素的值，也要存储数据元素之间的关系。

### 数据类型与抽象数据类型

数据类型是一个值的集合和定义在此集合上的一组操作的总称。

- 原子类型：其值不可再分的数据类型。
- 结构类型：其值可以再分解为若干成分（分量）的数据类型。

抽象数据类型*ADT*用数学化的语言定义数据的逻辑结构、定义运算。与具体的实现无关。

## 数据结构三要素

- 逻辑结构：元素之间的逻辑关系：线性结构、非线性结构（集合）、树结构、网状结构。

- 存储结构：数据在计算机中的表示（映像），也称物理结构：顺序存储结构（顺序表）、链式存储结构（链表）、索引存储（索引表）、散列存储（散列表、哈希表）。
- 数据运算：运算的定义是针对逻辑结构的，指出运算的功能；运算的实现是针对存储结构的，指出运算的具体操作步骤。

逻辑结构独立于存储结构，存储结构依托于逻辑结构。

## 算法

程序=数据结构+算法。算法为了处理信息。

是对特定问题求解步骤的一种描述，它是指令的有限序列，其中的每条指令表示一个或多个操作。

### 算法的特性

- 有穷性。
- 确定性。
- 可行性。
- 输入。零个或多个输入。
- 输出。一个或多个输出。

### 好算法的特点

- 正确性。
- 可读性。
- 健壮性。
- 高效率与低储量需求。

### 效率度量

#### *时间复杂度*

时间开销$T(n)$与问题规模$n$的关系。

当 n 规模够大时可以只考虑阶数高的部分。即$T(n) = O\big(T(n)\big)$。

- $T(n) = f(n) + g(n) = O\big(f(n)\big) + O\big(g(n)\big) = O\left(\max\big(f(n), g(n)\big)\right)$。多项相加只保留最高阶的项，系数为1。
- $T(n) = f(n) \times g(n) = O\big(f(n)\big) \times O\big(g(n)\big) = O\left(\max\big(f(n) \times g(n)\big)\right)$。多项相乘就都保留。
- $O(1) < O(\log_2 n) < O(n) < O(n\log_2 n) < O(n^2) < O(n^3) < O(2^n) < O(n!) < O(n^n)$。

如$T(n) = n^3 + n^2\log_2 n = O(n^3) + O(n^2\log_2 n) = O(n^3)$。

- 顺序执行的代码只会影响常数项，可以忽略。
- 循环就代表一个$n$，并列的循环不会叠加复杂度，只有嵌套的循环才会，嵌套的循环深度就是复杂度的幂数，具体的情况需要看循环内代码的逻辑分析。

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

这个代码的时间复杂度就是$O(\log_2 n)$。

```
// 假定数组中乱序存储 1 到 n 的整数
int array[n]={...};
void check(int array[], int n){
    for(int i=0; i<n; i++){
        if(array[i]==n){
            printf("%d", i);
            break;
        }
    }
}
check(array, n);
```

这个代码的执行时间依托于输入数据：

- 最好情况：元素$n$在第一个位置，$T(n) = O(1)$。
- 最坏情况：元素$n$在最后一个位置，$T(n) = O(n)$。
- 平均情况：$n$在任意一个位置概率相同，$T(n) = T\big((1+n)/2\big) = O(n)$。

在实际考虑时只会考虑最坏情况和平均情况。

*空间复杂度*

存储空间$S(n)$与问题规模$n$的关系。

算法原地工作是指算法所需的内存空间为常数级，即$O(1)$。

当$n$规模够大时可以只考虑阶数高的部分。即$S(n) = O\big(S(n)\big)$。

空间复杂度主要看程序所需要的变量所要的空间，一阶数组就是$n$，二阶就是$n$的二次方。

同时函数递归调用也会带来内存开销。当每一次递归调用的空间相等时，空间复杂度=递归调用的深度。

```
void check(int n){
    int array[n];
    if(n > 1) {
```

```
        check(n-1)
    }
}
check(n);
```

此时每一次的递归会使变量空间减去1，所以$S(n) = S\big((1+n)n/2\big) = O(n^2)$。

# 线性表

## 基本概念

### 逻辑结构

是具有相同数据类型的$n$个数据元素的有限序列。$n$表示表长。

$L = (a_1, a_2, \cdots, a_i, \cdots, a_n)$，其中$i$表示元素在线性表中的位序，从一开始。

- 存在唯一的第一个元素。
- 存在唯一的最后一个元素。
- 除第一个元素（表头元素）之外，每个元素均只有一个直接前驱。
- 除最后一个元素（表尾元素）之外，每个元素均只有一个直接后继。

### 物理结构

- 顺序存储结构：顺序表。
- 链式存储结构：链表。

## 顺序表

把逻辑上相邻的元素存储在物理位置上也相邻的存储单元中，元素之间的关系由存储单元的邻接关系来实现。$i$是元素$a_i$在线性表中的位序。

### 顺序表特点

1. 随机访问，可以在$O(1)$时间内找到对应元素。
2. 存储密度高，只用存储数据。
3. 拓展容量不方便。
4. 插入删除操作不方便。
5. 表中元素的逻辑地址与物理地址顺序相同。

### 顺序表定义

使用$C$语言的结构体定义顺序表，使用 `typedef` 定义一个 `ElemType` 表示数据基本类型，并定义最大长度 `MAXSIZE`：

```
// 初始化最大长度
#define MAXSIZE 25
```

```c
// 定义默认值
#define DEFAULTELEM 0
// 定义最大值
#define INFINITY 32767
// 定义默认数据类型
typedef char element_type;
```

可以使用静态分配空间：

```c
// 静态顺序表
typedef struct {
    element_type data[MAXSIZE];
    // 长度
    int length;
} StaticSequenceList;
```

也可以使用动态分配空间，动态分配空间还是顺序的，只不过可以替换原来空间：

```c
// 动态顺序表
typedef struct {
    // 给一个指针来分配动态数组
    element_type *data;
    // 已分配的最大容量
    int max_size;
    // 长度
    int length;
} DynamicSequenceList;
```

其中长度是指有数据的长度，而最大容量是指已经分配给动态数组的长度，插入时要考虑这个长度，不能溢出。

## 顺序表操作

### *顺序表初始化*

静态顺序表因为数组部分在创建时就已经设置好了，所以初始化就直接设置数据长度就可以了。

动态顺序表不仅需要设置数据长度与最大长度，还得分配数组初始空间。

### *顺序表增长数据空间长度*

只有动态顺序表才能增加。

### *顺序表插入*

倒序移动元素，最后将数据插入对应索引并长度加一。（这是一个较好的方式，因为如果插入的话其他元素会被挤住，倒序移动元素可以正好空出位置）

插入时间复杂度为：$T(n)=O(n)$，空间复杂度为$S(n)=O(1)$。

平均时间复杂度：假设$p_i$（$n_i=\frac{1}{n+1}$）是$i$位置上插入一个结点的概率，则在长度为$n$的线性表中插入一个结点时所需要移动结点的平均次数为$\sum_{i=1}^{n+1}p_i(n-i+1)=\sum_{i=1}^{n+1}\frac{1}{n+1}(n-i+1)=\frac{1}{n+1}\sum_{i=1}^{n+1}(n-i+1)=\frac{1}{n+1}\times\frac{n(n+1)}{2}=\frac{n}{2}$。

*顺序表删除*

正序移动元素并长度减一。

顺序表的删除时间复杂度为：$T(n)=O(n)$，空间复杂度为$S(n)=O(1)$。

*顺序表查找*

按位查找时间复杂度为$T(n)=O(1)$。

按值查找一般都是找到第一个元素等于指定值的元素，返回其位序，如果没有找到就返回$-1$。按位查找时间复杂度为$T(n)=O(n)$。

## 单链表

每个结点只包含一个指针域，也称为线性链表。

通常用头指针来标识一个单链表，如单链表$L$。

### 单链表特点

- 不要求大量连续空间，删除和插入方便。
- 不可随机存取。
- 要花费多余空间存放指针。

是非随机存取的存储结构。

### 单链表定义

使用 `LinkNode` 表示一个单链表结点的结构体，而使用 `LinkList` 表示一个单链表，其实 `LinkList` 是一个指向 `LinkNode` 的指针变量。如定义 `LinkList L` 等价于 `LinkNode* L`。

```
// 单链表结点
typedef struct LinkListNode {
  element_type data;
  struct LinkListNode* next;
} LinkListNode, *LinkList;
```

## 单链表操作

### *单链表初始化*

有带头节点与不带头节点的初始化的区别，带头节点代表第一个结点不存放数据，只是用于标识单链表的开始，但是区别不大，带头结点更好使用。

- 由于第一个数据结点的位置被存放在头结点的指针域中，因此在链表的第一个位置上的操作和在表的其他位置上的操作一致，无须进行特殊处理。
- 无论链表是否为空，其头指针都指向头结点的非空指针（空表中头结点的指针域为空)，因此空表和非空表的处理也就得到了统一。

### *单链表插入*

插入方式一共分为下面几种：

- 按位序插入：
  - 带头点结。
  - 不带头结点。
- 指定结点插入：
  - 前插入。
  - 后插入。

假定从第一个结点开始就是第0索引的结点。

带头结点的单链表头结点就是0号结点，不带头节点的第一个数据结点就是0号结点。

带头结点的单链表只能往头结点之后插入，所以插入索引必须从1开始。

头插法建立单链表：

- 每个结点的插入时间为$O(1)$，设单链表长为$n$，则总时间复杂度为$O(n)$。
- 实现了输入数据的就地逆置。

尾插法建立单链表

- 增设尾指针$r$。
- 生成的链表中结点数据与输入数据顺序一致。
- 总时间复杂度为$O(n)$。

插入有/无头节点单链表元素函数的后面代码可以使用后插入单链表元素函数来替代。

使用前插入的方法插入元素，可以使用头指针来得到整个链表信息，从而就能找到链表中的这个结点，但是如果没有头指针那么就无法实现了。且这种遍历的时间复杂度是$O(n)$。

还有另一种方式实现前插法，先后插一个元素，把前面结点的数据移动到这个新加的结点，把要新加的数据放在原来的结点，这就实现了后插，虽然地址没有变化，但是从数据上看就是前插，且时间复杂度是$O(1)$。

#### *单链表删除*

基本的方式和插入类似，都是转移$next$结点。

带头结点的也只能删除从1开始的结点，0的头结点不能删除。

时间复杂度为$O(n)$。

无头结点需要额外处理第一个结点

如果删除指定结点而不知道其前驱，也可以使用之前前插结点的方式，把该结点后继的结点的数据复制到本结点上，然后把后继结点删除，就相当于删除了本结点。时间复杂度为$O(1)$。

所以单链表还是不算方便。

#### *单链表查找*

按位查找时间复杂度为$O(n)$。

这样插入元素函数 `InsertLinkListWithHead` 只用 `GetLinkListNode(list,i-1)`和 `InsertNextLinkNode(p,elem)`两个函数完成。

#### *单链表建立*

可以使用尾插法建立单链表，从后面不断插入元素。需要定义一个尾指针来记录最后一位。

使用前插法建立单链表实际上也是使用后插操作，不过每一次后插的元素都是头结点，也不用使用尾指针。

前插法可以实现链表的逆置。

## 双链表

为了解决单链表只能单一方向扫描而无法两项遍历的缺点，使用了两个指针，`prior` 和 `next`，分别指向前驱和后继。

### 双链表定义

基本上与单链表的定义一致。

### 双链表操作

#### *双链表插入*

假如 p 的结点后要插入结点 s，则基本代码如下：

```
// 将插入的结点的后续接上原来的 p 的后续
s->next=p->next
// 将 p 后的结点的前驱连接到 s 上
p->next->prior=s;
// 将 s 的前驱连接到 p 上
s->prior=p;
// 将 p 的后继连接到 s 上
p->next=s;
```

操作上都是成对的，其中第一条第二条指令必须在最后一条指令之前，否则 p 的后继就会丢掉。

#### *双链表删除*

若删除结点 p 的后继结点 q：

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

## 循环链表

分为循环单链表和循环双链表。基本上变化不大。

原来的单链表的尾部指向 NULL，但是循环单链表的尾部是指向头部。

循环单链表即使没有头结点的地址，也可以通过循环得到整个单链表的信息。

从头到尾单链表需要遍历整个链表，而循环单链表只用移动一位就可以从头到尾。

循环双链表除此之外，头结点的 prior 指针还要指表尾结点（即某结点*p 为尾结点时，p->next==list）。

循环双链表为空表时，头结点的 prior 和 next 域都等于 list（即，指向自身）。

### 循环链表定义

循环链表和链表的结点定义是一致的。

## 静态链表

静态链表本质是一个数组，不过其内的基本元素不是基本数据类型而是结构体类型。

静态链表借助数组来描述线性表的链式存储结构，结点也有数据域和指针域，这里的指针是结点的相对地址（数组下标），又称游标。

静态链表和顺序表一样需要预先分配一块连续的内存空间。

数组0号元素充当链表的头结点且不包含数据。

如果一个结点是尾结点，其游标设置为$-1$。

具体的实现方式有多种，也可以0号元素数据保存头节点下标。

考的比较少。

### 静态链表特点

1. 增删改不需要移动大量数据元素。
2. 不能随机存取，只能从头节点开始。
3. 容量固定保持不变。

适用场景：

1. 不支持指针的低级语言。
2. 数据元素数量固定不变，如操作系统文件分配表 FAT。

### 静态链表定义

### 静态链表操作

#### *静态链表查找*

需要从头结点往后逐个遍历结点，时间复杂度为$O(n)$。

#### *静态链表插入*

如果要插入位序为 i，索引为 i-1 的结点：

1. 找到一个空结点（如何判断为空？可以先让 next 游标为某个特殊值如-2 等），存入数据元素。
2. 然后从头结点出发扎到位序为 i-1 的结点。
3. 修改新结点的 next。
4. 修改 i-1 位序结点的 next。

## 顺序表与链表对比

- 从逻辑结构来看，其都是线性结构的。
- 从物理结构来看，顺序表可以随机存取，存储数据密度高，但是分配与改变空间不变；链表空间离散，修改方便，但是不可随机存储，存储数据密度低。
- 从创建来看，顺序表需要申请一片大小适合的空间；而链表无所谓。

- 从销毁来看，顺序表需要将 length 设置为 0，从逻辑上销毁，再从物理上销毁空间，如果是静态分配的静态数组，系统会自动回收空间，而如果是动态数组，需要手动调用 `free` 函数；链表逐点进行 `free` 就可以了。
- 从增加删除来看，顺序表都要对后续元素进行前移或后移，时间复杂度为$O(n)$，主要来自于移动元素；而对于链表插入或删除元素只用修改指针就可以了，时间复杂度也为$O(n)$，主要来自于查找目标元素，但是链表的查找元素所花费的时间可能远小于移动元素的时间。
- 从查找来看，顺序表因为有顺序所以按位查找时间复杂度为$O(1)$，如果按值查找时间复杂度为$O(n)$，如果值是有序的则可以通过二分查找等方式降低在$O(\log_2 n)$的时间内找到；如果是链表的查找无论是按位还是按值都是$O(n)$的时间复杂度。

# 栈

栈结构与线性表类似，是只允许一端（表尾）进入或删除的线性表。即后进先出$LIFO$。

栈顶就是允许插入和删除的一端，而另一端就是栈底。

进栈顺序：$A \to B \to C \to D$，出栈顺序：$D \to C \to B \to A$。

如果有$n$个不同的元素进栈，出栈元素不同排列的个数为$\frac{1}{n+1}C_{2n}^{n}$，这就是卡特兰数。

## 顺序栈

### 顺序栈定义

设置栈顶指针可以为$0$（代表栈顶元素的下一个存储单元）也可以为$-1$（代表栈顶元素当前未知）。

### 顺序栈操作

#### *顺序栈初始化*

栈顶指针初始化为$-1$，因为索引最小为$0$。如果初始化为$0$也可以，不过其操作有所不同。

#### *进栈*

首先要判满，然后才能进栈。若栈顶指针指的是当前元素，即初值为$-1$，需要先自加再进栈，如果不先自加就会覆盖在原来的栈顶元素上。若栈顶指针指的是当前元素的下一个位置，即初值为$0$，则先进栈再自加，因为指向的下一个位置，所以指向的位置是空的，所以可以存入然后自加，若先自加则中间就空了一格。

#### *出栈*

首先要判空，然后才能出栈。若栈顶指针指的是当前元素，即初值为$-1$，需要先出栈再自减，如果由于指的是当前元素所以要先将这个指向的元素弹出，然后自减，否则弹出的就是靠近栈底的下一个元素。若栈顶指针指的是当前元素的下一个位置，即初值为0，则先自减再出栈，因为指向的下一个位置，所以指向的位置是空的，要先自减指向有元素的一格才能出栈。

对于出栈元素的处理，既可以将原来的存储单元设置为 NULL 也可以不处理，因为栈顶指针不指向这些单元，用户是不知道里面是什么的，之后重新用到这些存储单元也会覆盖原来的数据。

### 共享栈

即根据栈底不变，让两个顺序栈共享一个一维数组，将两个栈的栈底设在数组两端，栈顶向共享空间延申。

存取数据时间复杂度为$O(1)$。

## 链栈

链栈基本上就是只能操作一头的链表，所以从定义上其基本上没有区别。基本上以表头为栈顶。

## 栈的应用

### 括号匹配

即需要括号成双相对，且大小一样。

括号匹配时会发现最后出现的左括号最先被匹配$LIFO$。所以就可以通过栈来模拟这个匹配过程。

自左至右扫描表达式，若遇左括号，则将左括号入栈，若遇右括号，则将其与栈顶的左括号进行匹配，若配对，则栈顶的左括号出栈，否则出现括号不匹配错误，如果需要匹配但是栈空说明有单独的左或右括号，也匹配失败。如果结束，栈为空则正常结束，否则不匹配。

### 表达式求值

一般使用的都是中缀表达式，例如：$4+2\times3-10\div5$，按照运算法则，我们应当先算$2\times3$然后算$10\div5$，再算加法，最后算减法。

表达式分为三个部分：操作数、运算符、界限符。

如果不使用界限符，如中括号或小括号，可以使用后缀表达式（逆波兰表达式）或前缀表达式（波兰表达式）。

常用的中缀表达式是将运算符如加减乘除放在两个操作数中间，而后缀表达式就是放在两个操作数之后，前缀表达式就是放在两个操作数之前。

### *后缀表达式*

中缀转后缀的手算方法：

1. 确定中缀表达式中各个运算符的运算顺序进行排序。
2. 选择下一个运算符，按照**左操作数 右操作数 运算符**的方式组合一个个新的操作数。
3. 如果还有运算符没有处理就重复步骤二。

如$A+B*(C-D)-E/F$就是$ABCD-*+EF/-$和$ABCD-*EF/-+$。中缀转后缀的结果可以有不同的结果。

即使有不同的转换结果，但是如果我们要通过计算机实现这种转换算法，就必须保证算法的唯一性，所以规定后缀表达式中运算符的顺序就是中缀表达式中运算符生效的顺序，即结果是第一个而不是第二个。

所以后缀表达式转换必须遵循左优先的原则，能先计算左边的运算符就计算左边的运算符。

后缀表达式转换的程序实现：

1. 初始化一个栈，用于保存暂时不能确定运算顺序的运算符。
2. 从左到右处理每个元素。
3. 如果遇到操作数，直接加入后缀表达式。
4. 如果遇到界限符，如果遇到左括号直接入栈，如果遇到右括号依次弹出栈内运算符并加入到后缀表达式中，直到遇到新的左括号为止，不弹出左括号。
5. 遇到运算符，依次弹出栈中优先级高于或等于当前运算符的所有运算符并加入后缀表达式，若碰到左括号或栈空停止，之后再将当前运算符入栈。
6. 处理完所有字符后，将栈中剩余运算符依次弹出，并加入后缀运算符。

后缀表达式计算的程序实现：

1. 从左往右扫描下一个元素，直到处理所有元素。
2. 若扫描到操作数则压入栈，并回到1，若扫描到运算符则执行3。
3. 扫描到运算符则弹出两个栈顶元素，执行相应操作，运算结果压入栈，回到步骤一。
4. 先出栈的是右操作数，后出栈的是左操作数。

使用栈进行中缀表达式求值的程序实现：

1. 我们设定两个栈，一个用于存储运算符称之为运算符栈，另一个用于存储操作数称之为操作数栈。

2. 首先置操作数栈为空，表达式起始符“#”为运算符栈的栈底元素。
3. 依次读入表达式中每个字符，若是操作数则进操作数栈，若是运算符则和运算符栈栈顶元素比较优先级，若栈顶元素优先级高于即将入栈的元素，则栈顶元素出栈（优先级高的先出栈，再把优先级低的放进来），操作数栈弹出两个操作数和运算符一起进行运算，将运算后的结果放入操作数栈，直至整个表达式求值完毕（即运算符栈顶元素和要放入元素均为“#”）。

如果我们将后缀表达式转换为中缀表达式计算，可以从左往右扫描，每遇到一个运算符，就让运算符前面最近的两个操作数执行对应运算，合并为一个操作数。

### *前缀表达式*

中缀转后缀的手算方法：

1. 确定中缀表达式中各个运算符的运算顺序进行排序。
2. 选择下一个运算符，按照**运算符 左操作数 右操作数**的方式组合一个个新的操作数。
3. 如果还有运算符没有处理就重复步骤二。

遵循右优先原则，只要能计算右边就优先计算右边。

前缀表达式计算的程序实现：

1. 从右往左扫描下一个元素，直到处理所有元素。
2. 若扫描到操作数则压入栈，并回到 1，若扫描到运算符则执行 3.
3. 扫描到运算符则弹出两个栈顶元素，执行相应操作，运算结果压入栈，回到步骤一。
4. 先出栈的是左操作数，后出栈的是右操作数。

### 递归

- 递归表达式（递归体）。
- 边界条件（递归出口）。

函数调用的特点：最后被调用的函数最先执行结束$LIFO$。

函数调用时需要一个栈存储：调用返回地址、实参、局部变量。用栈来让递归算法转换为非递归算法。

递归可以将原始问题拆分为属性相同、规模较小的问题。但是如果太多层会造成栈溢出。

### 迷宫问题

- 思想：以栈$S$记录当前路径，则栈顶中存放的是“当前路径上最后一个位置信息”。
- 若当前位置“可通”，则纳入路径（入栈），继续前进。

- 若当前位置“不可通”，则后退（出栈），换方向继续探索。
- 若四周“均无通路”，则将当前位置从路径中删除出去。

# 队列

队列是只允许一端进行插入（入队或进队），一端进行删除（出队或离队）的线性表。即先进先出$FIFO$。

队列允许插入的一端就是队尾，允许删除的一端就是队头。

队列和栈是逻辑结构和物理结构没有不同，只是操作方式不同。

## 顺序队列

分配一块连续的存储单元存放队列中的元素，并附设两个指针，队头指针和队尾指针，队头指针指向队头元素，队尾指针指向队尾元素的下一个位置。

### 顺序队列操作

#### *顺序队列插入*

如果出队，则前面的空间会空闲，但是假如队尾指针会依照插入而不断加1，则我们的队尾指针最后会指向最后一个区域，计算机不知道前面是怎么样，所以就认为空间已经满了，实际上没有。这就是假溢出。

#### *顺序队列删除*

当如果我们必须保证所有的存储空间被利用，可以定义一个$size$表明队列当前的长度，就可以完全利用所有空间。

## 循环队列

对于顺序队列假溢出的解决的方法就是使用模运算，将队尾指针不仅仅是加一，而是加一后再取整个静态数组大小$MAXSIZE$的模，这样如果队列尾指针超过了范围也仍能回到最开始插入数据。这时候物理结构虽然是线性的，而逻辑上已经变成了环型的了。

所以与此同时，队列已满的条件也不再是队尾指针$= MAXSIZE$了，而是队尾指针的下一个指针是队头指针，这里最后一个存储单元是不能存储数据的，因为如果存储了数据那么头指针就等于尾指针，这在我们的定义中是空队列的意思。但是如何判断满队？

1. 牺牲最后的一个存储单元。入队少用一个队列单元。队满条件：$(rear + 1)\%MAXSIZE == front$，队空条件$front == rear$，从而队列元素个数$= (rear + MAXSIZE - front)\%MAXSIZE$。

2. 增设一个用来表示数据个数的成员。队满条件$length == MAXSIZE$，队空条件$length == 0$。队空和队满都是$front == rear$。
3. 可以定义一个$int$类型的$tag$，当进行删除操作就置$tag$为0，插入操作时置$tag$为1，只有删除才可能队空，只有插入才可能队满，所以就可以根据这个来判断。队空和队满都是$front == rear$。

### 链队

定义链队需要定义一个队头指针和一个队尾指针，队头指向队头结点，队尾指向队尾结点，即单链表最后一个结点，这跟顺序存储不同。

由于带头节点的链表对于出队比较简单，所以一般都定义为带头节点的链表。

### 双端队列

双端队列：只允许从两端插入、两端删除的线性表。

输入受限的双端队列：只允许从一端插入，两端删除的线性表。

输入受限的双端队列：只允许从一端删除，两端插入的线性表。

### 队列应用

#### 树的层次遍历

1. 根结点入队。
2. 若队空（所有结点都已处理完毕)，则结束遍历，否则重复三操作。
3. 队列中第一个结点出队，并访问之。若其有左孩子，则将左孩子入队；若其有右孩子，则将右孩子入队，返回二。

- 图的广度优先遍历。
- 进程争用$FCFS$策略。

#### 计算机系统

- 解决$CPU$与外设速度不匹配问题。
- 解决请求处理机问题。

## 数组

### 数组的定义

数组是由同类型的数据元素构成的有序集合，每个元素是数组元素，每个元素受$n$个线性关系的约束。其中每个元素在$n$个线性关系中的序号就是元素的下标，可以通过下标来访问元素。

数组是对线性表的推广。

数组一旦被定义其维数和维界就不能改变，所以数组只能对结构的初始化和销毁，以及元素的存取和修改。

所以数组的重点在于其存储。

## 数组的存储结构

### 一维数组

各数组元素大小相同，且物理上连续存放。

数组元素$a[i]$的存放地址=起始地址$LOC + i \times sizeof(ElemType)$。数组下标从0开始。

### 二维数组

二维数组存储方式还是同一维数组一样连续的。已知二维数组$b[M][N]$。

- 行优先：一行一行存储。$b[i][j]$的存储地址=起始地址$LOC + (i \times N + j) \times sizeof(ElemType)$。
- 列优先：一列一列存储。$b[i][j]$的存储地址=起始地址$LOC + (j \times M + i) \times sizeof(ElemType)$。

### 十字链表法

每个结点中包含行数、列数、元素值，以及两个指针，向下域指针$down$指向同第$j$列的下一个个元素，向右域指针$right$指向同第$i$行的下一个元素。

## 特殊数组的压缩

压缩存储指为多个值相同的元素只分配一个存储空间从而节省存储空间。

特殊矩阵是指具有许多相同矩阵元素或零元素，并且这些相同矩阵元素或零元素的分布有一定规律的矩阵。

若是索引都从1开始，则公式不发生改变，用$i-1$和$j-1$替换公式的$i$和$j$。。

### 对称矩阵

若对一个$n$阶方阵$A[0,n-1][0,n-1]$中的任意一个元素$a_{ij}$都有$a_{ij} = a_{ji}$，即主对角线对称元素相等的矩阵，就是对称矩阵。

其中元素可以分为上三角区域、主对角线和下三角区域，上下三角区域元素相等。所以可以将$A$存放在一维数组$B[n(n+1) \div 2]$中，从而$a_{ij} = b_k$，只存放下三角部分与主对角线部分元素。

对于$a_{ij}$而言，其在$i$行$j$列，第0行有1个元素，1行有两个元素，所以$i-1$行有$i$个元素，所以前$i-1$行共有$(1+i) \times i/2$个元素，最后加上第$i$行的$j$个元素。

从而$a_{ij}$对应的$k = 1 + 2 + \cdots + (i-1) + j = \frac{i(i+1)}{2} + j$。

- 当$i \geq j$时，即下三角区域与主对角线元素：$k = \frac{i(i+1)}{2} + j$。
- 当$i < j$时，即上三角区域：$k = \frac{j(j+1)}{2} + i$。

## 三角矩阵

### *下三角矩阵*

下三角矩阵指上三角区域的元素均为同一常量，其存储思想与对称矩阵一样，但是需要最后多一个存储空间存储上三角的常量。所以可以将$A$存放在一维数组$B[n(n+1) \div 2 + 1]$中。

- 当$i \geq j$时，即下三角区域与主对角线元素：$k = \frac{i(i+1)}{2} + j$。
- 当$i < j$时，即上三角区域：$k = \frac{n(n+1)}{2}$（最后一位）。

### *上三角矩阵*

上三角矩阵指下三角区域的元素均为同一常量，其存储思想与下三角矩阵一样，不过下标不同。

位于元素$a_{ij}\ (i \leq j)$前的元素个数有：第0行有$n$个元素，第1行有$n-1$个元素，第$i-1$行有$n-i+1$个元素，第$i$行有$j-i$个元素。

从而$a_{ij}$对应的$k = n + (n-1) + \cdots + (n-i+1) + (j-i) = \frac{i(2n-i+1)}{2} + (j-i)$。

- 当$i \leq j$时，即上三角区域与主对角线元素：$k = \frac{i(2n-i+1)}{2} + (j-i)$。
- 当$i > j$时，即下三角区域：$k = \frac{n(n+1)}{2}$。

### *三对角矩阵*

对角矩阵也称为带状矩阵。对于$n$阶方阵$A$的任意元素$a_{ij}$，当$|i-j| > 1$时，有$a_{ij} = 0$，则是三对角矩阵。

三对角矩阵除了以主对角线为中心的三条对角线的区域上的元素并不是完全为零外，其他元素都是零。

所以可以将三条对角线上的元素行优先地存储在一维数组中。第0行是两个元素，而1到$i-1$一共$i-1-1+1 = i-1$行每行三个元素，所以前$i-1$行一共$2 + (i-1) \times 3$个元素，第$i$一共$j-i+1$个元素。即$k = 2 + (i-1) \times 3 + j - i + 1$，所以得到$k = 2i + j$。

*稀疏矩阵*

若矩阵中非零元素很少，这个矩阵就是稀疏矩阵，若使用普通一维数组存储则十分浪费空间，所以一般只存储非零元素。

所以可以构成三元组(行标,列标,值)来存储。可以使用数组来存储也可以使用之前的十字链表法来存储。

稀疏矩阵压缩后就失去了随机存取的特性。

# 串

重点是字符串匹配模式，其他只做了解。

## 基本概念

- 串：零个或多个字符组成的有限序列。
- 子串：串中任意个连续的字符组成的子序列。
- 空串：长度为零的串。
- 空白串（空格串）：仅由一个或多个空格组成的串。
- 空串是任意串的子串，任意串是其自身的子串。

串的基本操作是对子串的操作。

## 串定义

### 顺序串

顺序串的结构定义方案

- 使用单独的变量$length$保存串长。
- 使用$data[0]$记录串长；使得字符位序与数组下标一致；但是由于$char$类型一个为一字节大小，所以能表示的数字是0到255，太大的串无法表示，大于的部分会被截断。
- 没有表示串长的变量，使用$\backslash 0$表示串结尾，对应$ASCII$码的0号字符。
- $data[0]$空余，使用单独的变量$length$保存串长，这个比较常用。
- 可以定长分配也可以用堆分配。

### 链串

如一般的链式存储结构定义一样，定义一个数据与指向下一位的指针。

但是如果你只在每个结点定义了一个字节的数据，但是又包含了四个字节的指针，那么存储利用率会很低。

如果是顺序表数据类型是整数类型，那么这种利用率低的情况确实无可奈何，但是对于串而言，因为一个字节存储一个字符，所以能一个字节存一个字符类型数据，所以为了提升数据存储利用率，可以每个结点存等多个字符。这个就是块链串。

## 模式匹配

模式匹配指在主串中找到与模式串相同的子串并返回其所在位置。

### 朴素模式匹配算法

从主串$T$、模式串$P$（子串）的第一个位置开始比较（$i=0,j=0$），若相等，则$i$，$j$各自$+1$，然后比较下一个字符。若不等，主串指针回溯到上一轮比较位置的下一个位置，子串回溯到0，再进行下一次比较。令子串长度为$m$，主串长度为$n$：

- 匹配成功的最好时间复杂度：$O(m)$：刚好第一个就匹配上了，总对比次数为子串长度。
- 匹配失败的最好时间复杂度：$O(n-m+1)=O(n-m)=O(n)$：匹配成功之前，每一个与第一个字符都匹配失败。
- 匹配失败的最坏时间复杂度：$O(nm-m^2+m)=O(nm)$：子串除了最后一个对不上，其余的都能对上，则每次遍历完一边后，又要走回头路；直到匹配成功/失败一共需要比较$m\times(n-m+1)$次。$m$：每次需要移动$m$次，$i$需要移动$n-m+1$次。

暴力匹配算法的最大问题就是对主串一位位进行对比，当后面的匹配失败后只能回溯主串，只移动一位重新匹配。

### KMP 算法

#### *原理*

$KMP$算法是对朴素模式匹配算法的优化。

朴素模式匹配算法的缺点就是当某些子串与模式串能部分匹配时，主串的扫描指针$i$经常回溯，从而导致时间开销。

主要思想是失配时，只有模式串指针回溯，主串指针不变，找到失配前模式串的最长公共前后缀并跳转到最大公共后缀开始匹配，且最大公共前后缀要小于左端子串长度。

那么如何理解？如主串是12345612346，模式串为12346，那么我们从1开始对比到5的时候失配。按照默认暴力匹配法不仅模式串要回溯到第一个，主串也要回溯到第二个即2重新对比。

但是我们一眼就能看出来2这个位置不需要进行对比，因为我们之前匹配过，字符串中只有最开始位置为1，其他位置都不为1，所以应该直接跳到没有对比的主串

位置进行对比，而不是重复对比之前的内容。这是我们大脑默认处理的过程，$KMP$就是模拟这个处理过程。

主串是未知的，而模式串是已知的，所以对于串匹配的优化必然基于模式串。

由于模式串在最开始就是已知的，所以在失配前主串和模式串必然相等，即我们可以选择模式串中能匹配的部分重新匹配，而不是直接从头开始。

### *公共前后缀*

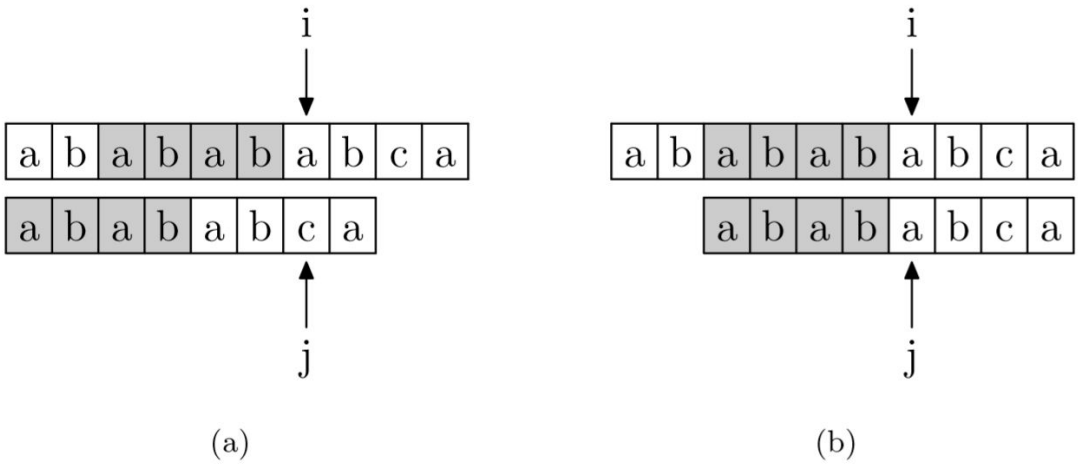
模式后滑动位数只与模式串本身的最大公共后缀有关，于主串无关。

- 前缀：对于字符串$A$，$B$，$A = B + S$，且$S$非空，则$B$是$A$的前缀。
- 后缀：对于字符串$A$，$B$，$A = S + B$，且$S$非空，则$B$是$A$的后缀。
- $PMT$值：前缀集合和后缀集合的交集中，最长元素的长度。
- 部分匹配表：$PMT$值集合，字符串所有前后缀的$PMT$值。

当一个位置失配时，那么子串前面的所有字符串都是配对的，所以对于子串前面的部分都是已知的了，需要从模式串的最开始开始对比，而一般的模式匹配要从主串的下一个重新开始匹配，但是如果我们找到了主串当前失配位置的前缀和后缀最大重合的地方，即公共前后缀，$PMT$值，就代表从这里开始就可以匹配了，前面的地方没必要匹配，可以直接多跳几步移动到公共后缀去开始重新匹配。

| 字符串 | 前缀 | 后缀 | 交集 | PMT |
|---|---|---|---|---|
| ‘a’ | ∅ | ∅ | ∅ | 0 |
| ‘ab’ | ‘a’ | ‘b’ | ∅ | 0 |
| ‘aba’ | ‘a’,‘ab’ | ‘ba’,‘a’ | ‘a’ | 1 |
| ‘abab’ | ‘a’,‘ab’,‘aba’ | ‘b’,‘ab’,‘bab’ | ‘ab’ | 2 |
| ‘ababab’ | ‘a’,‘ab’,‘aba’,‘abab’ | ‘a’,‘ba’,‘aba’,‘baba’ | ‘a’,‘aba’ | 3 |

所以字符串’$ababa$’的部分匹配值为00123，即可以得到部分匹配值表。




*KMP 算法*

如图可知在$a$和$c$处失配，由于前面是匹配的，所以可以直接对齐比较。

*next 数组*

失配移动位数$move$=已匹配字符数$(j-1)$-对应的部分匹配值$PM[j-1]$。（从而跳到开始有重复公共前缀的地方）

部分匹配值表就是子串应该跳转的索引值。当这个位失配，则子串应该跳转的索引值是失配位置前一位的$PM$值。

因为纯匹配值表要看前一位的值，所以可以把匹配表数据全部右移一位，这就可以直接看失配位置的表值了，定义为$next$。最开始的一位用$-1$表示，最后一位丢弃。

所以$move=(j-1)-next[j]$。即移动位数=匹配位数-本位的跳转值。

所以相当于子串的比较指针$j$回到$j=j-move=j-\big((j-1)-next[j]\big)=next[j]+1$。

所以$next$也可以全部加$1$，即得到子串变化表达式$j=next[j]$。$next$此时就是$j$失配时应该跳转到的索引值。

所以当$j=0$时，恒定$next[0]=0$（主串加一）$next[1]=1$。因为只有一个字母没有前一位所以是$-1$，只有两个字母前一位只有一个字母没有前后缀。（这里默认$next$数组从$0$开始，如果从$1$开始则索引全部加一）

1. 求$next[j+1]$，则已知前面的所有$next$表值$next[1],next[2]\cdots next[j]$。
2. 假设数组值$next[j]=k_1$（跳转索引），则有$P_1\cdots P_{k_1-1}=P_{j-k_1+1}\cdots P_{j-1}$（前$k_1-1$位字符与后$k_1-1$位字符重合）。

3. 如果$P_{k_1} = P_j$（即最后一位也一样，则得到在之前匹配基础上的更长的公共前后缀），则$P_1 \cdots P_{k_1-1}P_{k_1} = P_{j-k_1+1} \cdots P_{j-1}P_j$，则$next[j+1] = k_1 + 1$，否则进入下一步。
4. 假设$next[k_1] = k_2$、则有$P_1 \cdots P_{k_2-1} = P_{k_1-k_2+1} \cdots P_{k1-1}$。
5. 第二第三步联合得到$P_1 \cdots P_{k_2-1} = P_{k_1-k_2+1} \cdots P_{k1-1} = P_{j-k_1+1} \cdots P_{k_2-k_1+j-1} = P_{j-k_2+1} \cdots P_{j-1}$，即四段重合。
6. 这时候．再判断如果$P_{k_2} = P_j$，则$P_1 \cdots P_{k_2-1}P_{k_2} = P_{j-k_2+1} \cdots P_{j-1}P_j$，则$next[j+1] = k_2 + 1$，否则再取$next[k_2] = k_3$回到四。
7. 如果遇到0还没有结果，则表示前面的全部不重合，赋值为$0 + 1 = 1$。

即要计算当前位置的$next$值，就看前一位的$next$值所代表的索引指向的字符是否与前一位的字符相等，若相等，则是前一位的$next$值加一，若不等，则继续看前一位的$next$值指向的字符的$next$指向的字符与前一位字符是否相等，若相等则结果就是这个$next$值加一，否则继续按照$next$索引向前寻找。

### *KMP 匹配*

$KMP$算法在形式上跟简单的模式匹配算法类似，唯一不同的是当失配时指针$i$不动（主串不动）指针$j$回到$next[j]$的位置重新比较，当$j = 0$时$ij$同时加一，即主串第$i$个位置与模式串第一个字符不等时应该从主串$i + 1$个位置开始匹配。

### *算法性能*

使用$KMP$算法时需要先计算不同模式串$P$的$next$数组，时间复杂度为$O(m)$，然后使用$KMP$算法计算，时间复杂度为$O(n)$，从而平均时间复杂度为$O(m + n)$。

虽然普通模式匹配算法复杂度$O(mn)$，但是一般情况下接近于$O(m + n)$。

$KMP$算法对于重复部分比较多的模式串匹配效果更好。

### KMP 算法优化

$KMP$算法的$next$数组存在一定问题，当当前索引的值匹配失败，那么模式串的其他同样值的地方也一定会匹配失败。

如$goolggoogle$匹配$google$，其中$PMT$表格为：

| 序号 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 模式串 | g | o | o | g | l | e |
| next[j] | 0 | 1 | 1 | 1 | 2 | 1 |

其中匹配到第四个$l$时与$g$不匹配，按照表格会跳转匹配到第1个字符，但是由于序号1的字符也是$g$，所以这次跳转就是个浪费的对比。

所以可以直接将模式串所有相同值的部分的$next$值全部取为其$next$值对应索引的$next$值。

所以需要再次递归，将$next[j]$变为$next\left[next[j]\right]$直到两者不相等，令更新后数组为$nextval$。

| 序号 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 模式串 | g | o | o | g | l | e |
| next[j] | 0 | 1 | 1 | 1 | 2 | 1 |
| nextval[j] | 0 | 1 | 1 | 0 | 2 | 1 |

对于多个字符重复的字符串，则$nextval$的优化程度会更高：

| 序号 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 模式串 | a | a | a | a | b |
| next[j] | 0 | 1 | 2 | 3 | 4 |
| nextval[j] | 0 | 0 | 0 | 0 | 4 |

# 树

## 基本概念

### 树的基本概念

- 树：$n$个结点的有限集（树是一种递归的数据结构，适合于表示具有层次的数据结构）。是递归定义的。
- 根结点：只有子结点没有父结点的结点。除了根结点外，树任何结点都有且仅有一个前驱。
- 分支结点：有子结点也有父结点的结点。
- 叶子结点：没有子结点只有父结点的结点。
- 祖先：根结点到结点的路径上的任意结点都是该结点的祖先。
- 双亲：靠近根结点且最靠近该结点的结点。
- 兄弟：有共同双亲结点的结点。
- 堂兄弟：双亲结点在同一层的结点。
- 空树：结点数为0的数。
- 子树：当$n > 1$时，其余结点可分为$m$个互不相交的有限集合，每个集合本身又是一棵树，其就是根结点的子树。
- 结点的度：一个结点的孩子（分支）个数。
- 树的度：树中结点的最大度数。
- 结点的层次（深度）：从上往下数。

- 结点的高度：从下往上数。
- 树的高度（深度）：多少层。
- 两结点之间的路径：由两个结点之间所经过的结点序列构成。
- 两结点之间的路径长度：路径上所经过的边的个数。
- 树的路径长度：指树根到每个结点的路径长的总和，根到每个结点的路径长度的最大值是树的高。
- 有序树：树各结点的子树从左至右有次序不能互换。
- 无序树：树各结点的子树从左至右无次序可以互换。

### 森林的基本概念

- 森林时$m$棵互不相交的树的集合。
- 一颗树可以被分为森林。

### 树的性质

- 结点数=总度数+1。（加一是因为根结点）
- 树的度$m$代表至少一个结点度是为$m$，且一定是非空树，至少有$m+1$个结点；而$m$叉树指所有结点的度都小于等于$m$，可以是空树。
- 度为$m$的树以及$m$叉树第$i$层至多有$m^{i-1}$个结点。（如完全二叉树）
- 高度为$h$的$m$叉树至多有$\frac{m^h-1}{m-1}$个结点。
- 高度为$h$的$m$叉树至少有$h$个结点，度为$m$的树至少有$h+m-1$个结点。
- 具有$n$个结点的$m$叉树最小高度为$\lceil \log_m(n(m-1)+1) \rceil$。已知高度最小时所有结点都有$m$个孩子，所以$\frac{m^{h-1}-1}{m-1} < n <\leq \frac{m^h-1}{m-1}$，从而得到$h-1 < \log_m(n(m-1)+1) \leq h$。

## 二叉树

### 二叉树的基本概念

- 二叉树是$n$个结点构成的有限集合。
- 二叉树可以为空二叉树，也可以是由一个根结点和两个互不相交的被称为根的左子树和右子树构成。左子树和右子树又分别是一棵二叉树，左右子树不能颠倒。
- 满二叉树：一棵高度为$h$，含有$2^h-1$个结点的二叉树；只有最后一层有叶子结点，不存在度为1的结点；按层序从1开始编号，结点$i$的左孩子为$2i$，右孩子为$2i+1$，父结点如果有为$\lfloor \frac{i}{2} \rfloor$。
- 完全二叉树：当且仅当其每个结点都与高度$h$满二叉树编号1到$n$的结点一一对应时该二叉树就是完全二叉树；只有最后两层有叶子结点，最多只有一个度为1的结点，即$n_1=0/1$，且一定为左孩子；$i \leq \lfloor \frac{n}{2} \rfloor$为分支结点，$i > \lfloor \frac{n}{2} \rfloor$为叶子结点。

- 二叉排序树：左子树上所有结点的关键字均小于根结点的关键字；右子树上所有结点的关键字均大于根结点的关键字；左右子树又各是一棵二叉排序树。
- 平衡二叉树：树上任一结点的左子树和右子树的深度之差不超过1。

## 二叉树的性质

- 设非空二叉树中度为0、1和2的结点个数分别为$n_0$，$n_1$、$n_2$，则$n_0 = n_2 + 1$（叶子结点比二分结点多一个）。假设树中结点的总数为$n$，则$n = n_0 + n_1 + n_2$，又根据树的结点等于总度数+1得到$n = n_1 + 2n_2 + 0n_0 + 1$，所以相减就得到结论。
- 二叉树的第$i$层至多有$2^{i-1}$个结点。
- 高度为$h$的二叉树至多有$\frac{m^h-1}{m-1}$个结点。
- 具有$n$个结点的完全二叉树的高度$h = \lceil \log_2(n+1) \rceil$或$\lfloor \log_2 n \rfloor + 1$。$2^{h-1} - 1 < n \leq 2^h - 1$。
- 完全二叉树最多只有一个度为1的结点，度为0度为2的结点的个数和一定为奇数；若完全二叉树有$2k$个结点，则必然$n_1 = 1$，$n_0 = k$，$n_2 = k - 1$，若完全二叉树有$2k - 1$个结点，则必然$n_1 = 0$，$n_0 = k$，$n_2 = k - 1$。

## 二叉树存储结构

### *顺序存储*

如果是完全二叉树，可以按照顺序进行存储，如果$i$有左孩子则$2i \leq n$，若有右孩子则$2i + 1 \leq n$，若有叶子或分支结点则$i > \lfloor \frac{n}{2} \rfloor$。

如果不是完全二叉树，则让二叉树的编号与完全二叉树相对应再存入数组，其他的结点为空，这种存储方法会浪费较多内存，最坏情况下高度为$h$，且只有$h$个结点的单支树也需要$2^h - 1$个存储单元。

这种存储结构需要从下标1开始存储，若从0开始则不满足父子结点的性质。

### *链式存储*

链式树具有两个分别指向左右子树的指针。

在含有$n$个结点的二叉链表中，含有$n + 1$个空链域。

含有$n$个结点的二叉链表中，链域一共有$2n$个（每个点有两个链域）。对于除了根结点以外的每个点都是有一个父亲结点，所以一共有$n - 1$个指针指向某个结点，于是形成$n - 1$个有内容的链域（减1即是根结点）所以一共有$2n - (n - 1) = n + 1$个链域没有指向任何东西。

如果要保存父结点的位置，可以添加一个父结点指针，从而变成三叉链表。

## 二叉树遍历

遍历是按照某种次序将所有结点都访问一遍。

### *顺序遍历*

顺序遍历就是深度优先的遍历，分为三种：

- 先序遍历：根左右$NLR$。
- 中序遍历：左根右$LNR$。
- 后序遍历：左右根$LRN$。

根据算算数表达式的分析树的不同先序、中序、后序遍历方式可以得到前缀、中缀、后缀表达式。

若树的高度为$h$，则时间复杂度为$O(h)$。

### *递归与非递归*

借助栈可以将本来是递归算法的顺序遍历变为非递归方式。

如中序遍历：

1. 沿着根的左孩子结点依次入栈，直到左孩子为空。表示找到了最左边的可以输出的结点。
2. 栈顶元素出栈并访问。
3. 若栈顶元素的右孩子为空，则继续执行步骤二。
4. 若栈顶元素的右孩子不为空，则对其右子树执行步骤一。

先序遍历与中序遍历类似，只是第一步就需要访问中间结点。

后序非递归遍历算法的思路：从根结点开始，将其入栈，然后沿其左子树一直往下搜索，直到搜索到没有左孩子的结点，但是此时不能出栈并访问，因为如果其有右子树，还需按相同的规则对其右子树进行处理。直至上述操作进行不下去，若栈顶元素想要出栈被访问，要么右子树为空，要么右子树刚被访问完（此时左子树早已访问完），这样就保证了正确的访问顺序。

### *层序遍历*

层序遍历就是广度优先的遍历。

1. 初始化一个辅助队列。
2. 根结点入队。
3. 若队列非空，则队头结点出队，访问该结点，如果有并将其左右孩子入队。
4. 重复步骤三直至队列空。

## 遍历序列构造二叉树

若只给出一棵二叉树的前/中/后/层序遍历序列中的一种不能唯一确定一棵二叉树。只有给出中序遍历序列才可能推出唯一二叉树，因为无法确定根结点相对于左右结点的位置：

- 前序+中序：
- 后序+中序。
- 层序+中序。

### *前序+中序*

前序：根+左+右；中序：左+根+右。所以根据三个部分对应相同可以推出。

先序遍历的第一个结点一定是二叉树根结点。中序遍历中根结点必然将序列分为两个部分，前一个序列是左子树的中序序列，后一个序列是右子树的中序序列。同理先序序列中子序列的第一个结点就是左右子树的根结点。

### *后序+中序*

后序：左+根+右；中序：左+根+右。所以根据三个部分对应相同可以推出。

### *层序+中序*

层序：根+左根+右根；中序：左+根+右。所以根据根结点和左右子树的根结点来确定。

## 线索二叉树

对于二叉树的遍历，只能从根结点开始遍历，如果给任意一个结点是无法完成遍历的。一般的二叉树的左右结点是用来表示父子关系，而不能体现遍历关系。

所以我们就想能否保存结点的前驱和后继，从而能减少重复遍历树。因为一棵树很多结点的左右结点可能是空的，那么这些空闲的指针可以不代表左右子树的根结点，而是用来表示当前遍历方法的前驱或后继。当这个指针表示的是前驱或后继就称为线索，指向前驱的就是前驱线索，由左孩子指针担当，指向后继的就是后继线索，由右孩子指针担当。

### *线索化*

线索化就是要遍历一遍二叉树，然后对当前结点进行处理。

为了区分其左右孩子指针是指向什么，要在结点中新建两个$tag$位，如当$ltag = 0$表示$lchild$指向的是左孩子结点，而为1表示其指向前驱。

- 确定线索二叉树类型——中序、先序或后序。
- 按照对应遍历规则，确定每个结点访问顺序并写上编号。

- 将$n+1$个空链域连上前驱后继。
- 没有前驱或后继就指向 NULL。
- 这种结构称为线索链表。

在先序线索化的时候要注意，由于是根左右的顺序，在访问根结点时候进行线索化可能就会将左孩子结点由指向左孩子变成指向前驱的线索（该结点本来就没有左孩子），然后处理左子树时会跳到这里指向前驱的线索即前一个结点，就会不断在这里循环（程序把前驱当作左孩子不断回撤）。所以在先序遍历二叉树时要根据$ltag$值判断是否是前驱线索再进行遍历左子树。而右孩子结点则不会有这个问题，因为访问顺序必然是左右，所以不管二叉树右孩子结点指向的是右孩子还是后继都是在当前访问结点后应该访问的结点。

而中序线索化和后序线索化都没有这种问题，因为当前结点的前驱在此时按处理顺序都已经处理完了。

同时三种线索化都需要处理最后一个结点，当最后一个结点的右孩子指针为 NULL，要$pre->rtag=1$。

*查找前驱后继*

如果某结点的左右孩子指针有孩子而不是指向前驱后继，那么怎么找其前驱后继？

- 中序线索二叉树中找到结点$*P$的中序后继$next$：
  - 若$p$右孩子指针指向后继：$p->rtag==1$，则$next=p->rchild$。
  - 若$p$右孩子指针指向右子树根结点：$p->rtag==0$，则$next=p$右子树中最左下结点。
  - 所以可以利用线索对二叉树实现非递归的中序遍历。
- 中序线索二叉树中找到结点$*P$的中序前驱$pre$：
  - 若$p$左孩子指针指向前驱：$p->ltag==1$，则$pre=p->lchild$。
  - 若$p$左孩子指针指向左子树根结点：$p->ltag==0$，则$pre=p$左子树中的最右下结点。
  - 所以可以利用线索对二叉树实现非递归的逆向中序遍历。
- 先序线索二叉树中找到结点$*P$的先序后继$next$：
  - 若$p$右孩子指针指向后继：$p->rtag==1$，则$next=p->rchild$。
  - 若$p$右孩子指针指向右子树根结点：$p->rtag==0$，如果$p$有左孩子，则$p->next=p->lchild$，如果$p$没有左孩子，则肯定有右孩子，$p->next=p->rchild$。
  - 所以可以利用线索对二叉树实现非递归的先序遍历。
- 先序线索二叉树中找到结点$*P$的先序前驱$pre$：
  - 若$p$左孩子指针指向前驱：$p->ltag==1$，则$pre=p->lchild$。
  - 若$p$左孩子指针指向左子树根结点：$p->ltag==0$，先序遍历中左右子树的根结点只可能是后继，必须向前找。

  - 如果没有父结点所以这时候就找不到$p$的前驱，只能从头开始先序遍历。
  - 如果有父结点，则又有四种情况：
    - $p$为左孩子，则根据根左右，$p$的父结点为根所以在$p$的前面，$p->pre = p->parent$。
    - $p$为右孩子，其左兄弟为空，则根据根左右，顺序为根右，所以$p->pre = p->parent$。
    - $p$为右孩子且有左兄弟，根据根左右，$p$的前驱就是左兄弟子树中最后一个被先序遍历的结点，即在$p$的左兄弟子树中优先右子树遍历的底部。
    - 若$p$是根结点，则没有先序前驱。
- 后序线索二叉树中找到结点$*P$后序后继$next$：
  - 若$p$右孩子指针指向后继：$p->rtag == 1$，则$next = p->rchild$。
  - 若$p$右孩子指针指向右子树根结点：$p->rtag == 0$，则根据左右根顺序，左右孩子结点必然是$p$的前驱而不可能是后继，所以找不到后序后继。
  - 如果没有父结点只能使用从头开始遍历的方式。
  - 如果有父结点则又有四种情况：
    - $p$为右孩子，根据左右根，所以$p->next = p->parent$。
    - $p$为左孩子，右孩子为空，根据左右根，所以$p->next = p->parent$。
    - $p$为左孩子，右孩子非空，根据左右根，所以$p->next$ =右兄弟子树中第一个被后序遍历的结点，即右子树优先左兄弟子树遍历的底部。
    - 若$p$是根结点，则没有后序后继。
- 后序线索二叉树中找到结点$*P$后序前驱$pre$：
  - 若$p$左孩子指针指向前驱：$p->ltag == 1$，则$pre = p->lchild$。
  - 若$p$左孩子指针指向左子树根结点：$p->ltag == 0$，则又有两种情况：
    - 若$p$有右孩子，则按照左右根的情况遍历，右在根的前面，所以$p->pre = p->rchild$。
    - 若$p$没有右孩子，按照左根的顺序，则$p->pre = p->lchild$。

## 树与森林

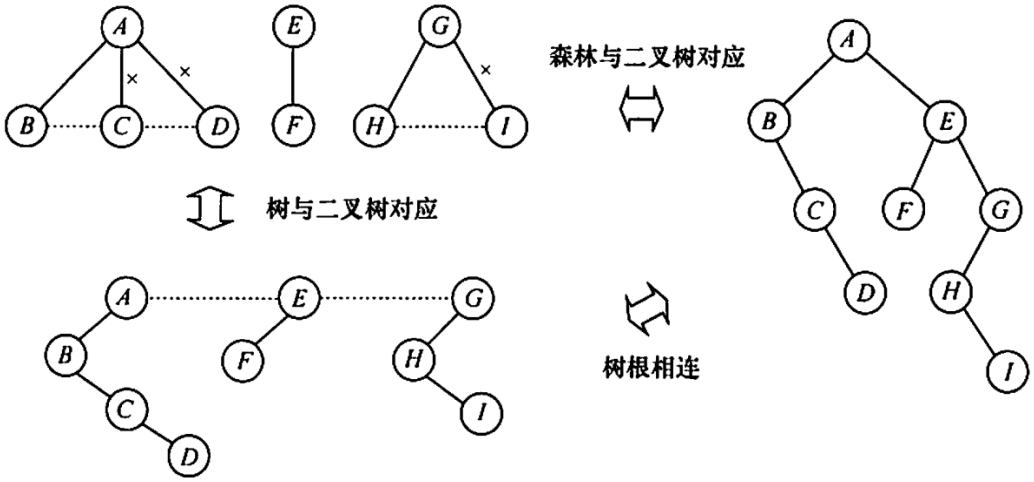
### 树的存储结构

- 双亲表示法：是一种顺序存储方式，一般采用一组数组，每个结点中保存指向双亲的伪指针。查找双亲方便，但是查找孩子就只能从头遍历。

- 孩子表示法：是顺序加链式存储方法，顺序存储所有元素，添加一个$firstChild$域，指向第一个孩子结构体的指针，孩子结构体包括元素位置索引与指向下一个孩子结构体的$next$指针。寻找孩子比较方便，但是寻寻找双亲需要遍历$n$个结点$n$个孩子链表。
- 孩子兄弟表示法：是一种链式存储方式，定义了两个指针，分别指向第一个孩子与右兄弟，类似于二叉树，可以利用二叉树来实现对树的处理。也称为二叉树表示法。可以将树操作转换为二叉树的操作，但是查找双亲麻烦。可以为每个结点设置一个指向双亲的结点。

### 森林与树的转换

树与森林的转换，树与二叉树的转换都可以使用孩子兄弟表示法来实现，左孩子右兄弟，如果是森林则认为其根结点为兄弟。




*树*

#### *树转换为二叉树*

树转换为二叉树的规则：每个结点左指针指向它的第一个孩子，右指针指向它在树中的相邻右兄弟，这个规则又称“左孩子右兄弟”。由于根结点没有兄弟，所以对应的二叉树没有右子树。

树转换成二叉树的画法：

1. 在兄弟结点之间加一连线。
2. 对每个结点，只保留它与第一个孩子的连线，而与其他孩子的连线全部抹掉。
3. 以树根为轴心，顺时针旋转45°。

#### *森林转换为二叉树*

将森林转换为二叉树的规则与树类似。先将森林中的每棵树转换为二叉树，由于任何一棵和树对应的二叉树的右子树必空，若把森林中第二棵树根视为第一棵树根的

右兄弟，即将第二棵树对应的二叉树当作第一棵二叉树根的右子树，将第三棵树对应的二叉树当作第二棵二叉树根的右子树……以此类推，就可以将森林转换为二叉树。

森林转换成二叉树的画法：

1. 将森林中的每棵树转换成相应的二叉树。
2. 每棵树的根也可视为兄弟关系，在每棵树的根之间加一根连线。
3. 以第一棵树的根为轴心顺时针旋转45°。

### *二叉树转换为森林*

二叉树转换为森林的规则：若二叉树非空，则二叉树的根及其左子树为第一棵树的二叉树形式，故将根的右链断开。二叉树根的右子树又可视为一个由除第一棵树外的森林转换后的二叉树，应用同样的方法，直到最后只剩一棵没有右子树的二叉树为止，最后再将每棵二叉树依次转换成树（左边是孩子右边是兄弟还原），就得到了原森林。

## 树的遍历

- 先根遍历：若树非空，先访问根结点，再依次对每棵子树进行先根遍历。
- 后根遍历：若树非空，先依次对每棵子树进行后根遍历，最后访问根结点。
- 层次遍历：用辅助队列实现：
    1. 若树非空，根结点入队。
    2. 若队列非空，队头元素出队并访问，同时将该元素的孩子依次入队。
    3. 重复步骤二直到队列为空。

## 森林的遍历

先序遍历森林：

1. 访问森林中第一棵树的根结点。
2. 先序遍历第一棵树中根结点的子树森林。
3. 先序遍历除去第一棵树之后剩余的树构成的森林。

如之前图中森林先序遍历为$ABCDEFGHI$。

中序遍历森林：

1. 先序遍历第一棵树中根结点的子树森林。
2. 访问森林中第一棵树的根结点。
3. 中序遍历除去第一棵树之后剩余的树构成的森林。

可以把每个树先按序遍历再合在一起，也可以先转换为二叉树再遍历。

如之前图中森林序中遍历为$BCDAFEHIG$。

如果通过转换，那么遍历的结果是等价的：

| 树 | 森林 | 二叉树 |
| --- | --- | --- |
| 先根遍历 | 先序遍历 | 先序遍历 |
| 后根遍历 | 中序遍历 | 中序遍历 |

### 转换关系

假设森林为$F$，树为$T$，转换而来的二叉树为$B$。

#### *结点关系*

- $T$有$n$个结点，叶子结点个数为$m$，则$B$中无右孩子的结点个数为$n-m+1$个。

树转换为二叉树时，树的每个分支节结点的所有子结点的最右子结点无右孩子，根结点转换后也无右孩子。

- $F$有$n$个非终端结点，则$B$中无右孩子的结点有$n+1$个。

根据森林与二叉树转换规则“左孩子右兄弟”，$B$中右指针域为空代表该结点没有兄弟结点。森林中每棵树的根结点从第二个开始依次连接到前一棵树的根的右孩子，因此最后一棵树的根结点的右指针为空，这里有一个。另外，每个非终端结点即代表有孩子，其所有孩子结点不论有多少个兄弟，在转换之后，最后一个孩子的右指针一定为空，故树$B$中右指针域为空的结点有$n+1$个。

#### *边关系*

- $F$有$n$条边、$m$个结点，则$F$包含$T$的个数为$m-n$。

若有$n$条边，则如果全部组成最小的树每个需要两个结点，总共需要$2n$个结点，组成$n$根树。假定$2n>m$，则还差$2n-m$个结点才能两两成树，所以少的这些结点不能单独成树，导致有$2n-m$个结点只能跟其他现成的树组成结点大于二的树。所以此时只能组成$n-(2n-m)=m-n$棵树。

## 树的应用

### 哈夫曼树

#### *哈夫曼树的定义*

- 路径和路径长度：从树中的一个结点到另一个结点之间的分支构成这两个结点之间的路径，路径上的分支数目称作路径长度。
- 结点的权：有某种现实含义的数值。
- 结点的带权路径长度：从根到该结点的路径长度（经过边数）与该结点权的乘积称为结点的带权路径长度。

- 树的带权路径长度：树中所有**叶子**的带权路径长度之和称为树的带权路径长度$WPL = \sum_{i=1}^{n} w_i\, l_i$。
- 哈夫曼树（最优二叉树）：带权路径长度最短的二叉树。不一定是完全二叉树。
- 哈夫曼树不存在度为1的结点。

### *构造哈夫曼树*

给定$n$个权值分别为$w_1, w_2 \cdots w_n$的结点，构造哈夫曼树的算法描述如下：

1. 将这$n$个结点分别作为$n$棵仅含一个结点的二叉树，构成森林$F$。
2. 构造一个新结点，从$F$中选取两棵根结点权值最小的树作为新结点的左、右子树，并且将新结点的权值置为左、右子树上根结点的权值之和。默认树较深的在右侧。
3. 从$F$中删除刚才选出的两棵树，同时将新得到的树加入$F$中。
4. 重复步骤二和三，直至$F$中只剩下一棵树为止。

- 每个初始结点最终都会变成叶子结点，且权值越小到根结点的路径长度越长。
- 哈夫曼树的结点总数为$2n - 1$。
- 构建哈夫曼树时，都是两个两个合在一起的，所以没有度为一的结点，即$n_1 = 0$。
- 哈夫曼树不唯一，但是$WPL$必然最优。

哈夫曼树适合采用顺序结构：已知叶子结点数$n_0$，且$n_1 = 0$，则总结点数为$2n_2 + 1$（或$2n_0 - 1$），且哈夫曼树构造过程需要不停地修改指针，用链式存储的话很容易造成指针偏移。

### *哈夫曼编码*

哈夫曼编码基于哈夫曼树，利用哈夫曼树对01的数据进行编码，来表示不同的数据含义，因为哈夫曼树必然权值最小，所以对于越常使用的编码越短，越少使用的编码越长，所以发送信息的总长度是最小的。

将编码使用次数作为权值构建哈夫曼树，然后根据左0右1的原则，按根到叶子结点的路径就变成了哈夫曼编码。

哈夫曼编码是可变长度编码，即允许对不同字符用不等长的二进制表示，也是一个前缀编码，没有一个编码是另一个编码的前缀。

同样哈夫曼编码也可以用于压缩。

## 并查集

将一个集合划分为互不相交的子集。类似森林。

一般用树或森林的双亲表示作为并查集的存储结构，每个子集用一个树表示。

用数组元素的下标表示元素名，用根结点的下标表示子合集名，根节点的双亲结点为负数。

查找，查找两个元素是否属于同一个集合。

合并，如果两个元素不属于同一个集合，且所在的两个集合互不相交，则合并这两个集合。

*存储结构*

如子集$S_1=\{A,B,D,E\}$、$S_2=\{C,H\}$、$S_3=\{F,G,I\}$。

存储结构为：

| 数据元素 | A | B | C | D | E | F | G | H | I |
|---|---|---|---|---|---|---|---|---|---|
| 数组下标 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| 双亲 | -1 | 0 | -1 | 0 | 3 | -1 | 5 | 2 | 5 |

*时间复杂度*

判断两个元素是否属于同一集合只需要找到其根节点进行比较。

查的时间复杂度为$O(n)$，并的时间复杂度为$O(1)$。

*应用*

1. 判断图的连通分量数—遍历各边，有边相连的两个顶点确认连通，“并”为同一个集合。只要是相互连通的顶点都会被合并到同一个子集合中，相互不连通的顶点一定在不同的子集合中。
2. $Kruskal$算法的最小生成树–各边按权值递增排序，依次处理：判断是否加入一条边之前，先查找这条边关联的两个顶点是否属于同一个集合（即判断加入这条边之后是否形成回郡），若形成回路，则继续判断下一条边；若不形成回路，则将该边和边对应的顶点加入最小生成树$T$，并继续判断 下一条边，直到所有顶点都已加入最小生成树$T$。

# 图

主要了解概念，算法具体实现不是重点。

## 基本概念

### 图的定义

图是顶点集和边集构成的二元组，即图$G$由顶点集$V$和边集$E$组成，记为$G=(V,E)$，其中$V(G)$表示图$G$中顶点的有限非空集，$E(G)$表示图$G$中顶点之间的关系（边）集合。

若$V=\{v_1,v_2\cdots,v_n\}$，则用$|V|$表示图$G$中顶点的个数，也称图$G$的阶，$E=\{(u,v)|u\in V,v\in V\}$，用$|E|$表示图$G$中边的条数。

图一定是非空的，即$V$一定是非空集。

## 图的类别

- 无向图：若$E$是无向边（简称边）的有限集合时，则图$G$为无向图。边是顶点的无序对，记为$(v,w)$或$(w,v)$，因为$(v,w)=(w,v)$，其中$v$、$w$是顶点。可以说顶点$w$和顶点$v$互为邻接点。边$(v,w)$依附于顶点$w$和$v$，或者说边$(v,w)$和顶点$v$、$w$相关联。
- 有向图：若$E$是有向边（也称弧）的有限集合时，则图$G$为有向图。弧是顶点的有序对，记为$<v,w>$，其中$v$、$w$是顶点，$v$称为弧尾，$w$称为弧头，$<v,w>$称为从顶点$v$到顶点$w$的弧，也称$v$邻接到$w$，或$w$邻接自$v$。$<v,w>\neq<w,v>$。
- 简单图：不存在重复边，且不存在顶点到自身的边。一般的图默认是简单图。
- 多重图：图$G$中某两个顶点之间的边数多于一条，又允许顶点通过同一条边与自己关联。
- 无向完全图：对于无向图$|E|\in[0,n(n-1)/2]$，无向图中任意两个顶点之间都存在边，即$|E|=n(n-1)/2$。
- 有向完全图：对于有向图$|E|\in[0,n(n-1)]$，有向图中任意两个顶点之间都存在方向相反的两条弧，即$|E|=n(n-1)$。
- 稀疏图：一般$|E|<|V|\log|V|$的图。
- 稠密图：一般$|E|>|V|\log|V|$的图。
- 树：不存在回路，且连通的无向图。（与图是逻辑的区别）
- 有向树：一个顶点的入度为0，其余顶点入度均为1的有向图。

## 顶点的度

### *度的定义*

对于无向图，顶点$v$的度是指依附于该顶点的边的条数，记为$TD(v)$。

对于有向图，入度是指以顶点$v$为终点的有向边的条数，记为$ID(v)$；出度指以顶点$v$为起点的有向边的条数，记为$OD(v)$；顶点$v$的度就是其入度和出度之和，即$TD(v)=ID(v)+OD(v)$。

有向树：一个顶点的入度为0，其他顶点入度均为1的有向图。

### *度的关系*

- 对于具有$n$个顶点，$e$条边的无向图，$\sum_{i=1}^{n}TD(v_i)=2|E|=2e$，即无向图的全部顶点的度的和等于边数的两倍。因为每条边都与两个顶点关联。
- 对于具有$n$个顶点，$e$条边的有向图，$\sum_{i=1}^{n}ID(v_i)=\sum_{i=1}^{n}OD(v_i)=|E|=e$。因为每条有向边都有一个起点和终点。

- 对于$n$个顶点的无向图，每个顶点的度最大为$n-1$。因为任意一个顶点可以与其他$n-1$个顶点相联。默认是简单图，即不能自己连向自己。
- 对于$n$个顶点的有向图，每个顶点的度最大为$2n-2$。因为任意一个顶点可以与其他$n-1$个顶点有指向相反的两条边。
- 无向连通图的每个顶点的度都是2。

### 顶点的关系

- 路径：从一个点到另一个点所经过的顶点序列。由顶点和相邻顶点序偶构成的边所形成的序列。
- 回路（环）：第一个顶点与最后一个顶点相同的路径。
- 长度（无权图）：沿路径所经过的边数成为该路径的长度。
- 简单路径：路径中的顶点不重复出现。
- 简单回路：由简单路径组成的回路。（除第一个和最后一个顶点外其余顶点不重复出现的回路）
- 点到点的距离：从顶点$u$到顶点$v$的最短路径若存在，则此路径的长度就是从$u$到$v$的路径，若不存在路径，则记该路径为无穷。

若一个图有$n$个顶点，有大于$n-1$条边，则此图一定有环。

### 图的连通

#### *连通概念*

- 连通：在无向图中，若从顶点$v$到顶点$u$有**路径**存在，则称$uv$是连通的。
- 强连通：在有向图中，若从顶点$v$到顶点$u$和从顶点$u$到顶点$v$之间都有**路径**（而不是弧），则称$uv$是强连通。
- 连通图：无向图中任意两个顶点之间都是连通的。
- 强连通图：有向图中任意两个顶点之间都是强连通的。
- 子图：设有两个图$G=(V,E)$和$G'=(V',E')$，若$V'$是$V$的子集，$E'$是$E$的子集，则$G'$是$G$的子图。
- 生成子图：若有满足$V(G')=V(G)$的子图$G'$，则$G'$是$G$的生成子图。（即子图包含所有顶点，但不一定包含所有的边）
- 连通分量：无向图$G$中的极大连通子图称为$G$的连通分量；对任何连通图而言，连通分量就是其自身。
- 强连通分量：有向图$G$中的极大连通子图称为$G$的强连通分量；对任何强连通图而言，强连通分量就是其自身。
- 生成树：包含连通图中全部顶点的一个极小连通子图。若图的顶点为$n$，则其生成树包含$n-1$条边，若去掉生成树的一条边则会变成非连通图，若加上一条边则会形成一个回路。
- 生成森林：在非连通图中，连通分量的生成树构成了非连通图的生成森林。

对于无向图：

- 极大连通子图：用来讨论图的连通分量，要求连通子图包含其所有的边。
  - 连通图的极大连通子图就是它本身。
  - 非连通图中有多个连通分量（不同的点相连从而连通），也就是可以有多个极大连通子图。
- 极小连通子图：用来讨论图的生成树，要保持图连通也要让边数最小的子图。
  - 极小连通子图只在无向图中才有。
  - 极小连通子图中包含图中全部的顶点（和极大不同，极大不要求包含所有的顶点）。
  - 用边将极小连通图中的所有边都连接起
  - 极小连通子图和生成树的概念不是等价的，生成树是包含图中全部顶点的一个极小连通子图。

对于有向图：

- 极大强连通子图：
  - 强连通图的极大强连通子图为其本身。（是唯一的）
  - 非强连通图有多个极大强连通子图。（非强连通图的极大强连通子图叫做强连通分量）
- 不存在极小强连通子图的概念，因为树没有方向性。

无向图研究连通性，有向图研究强连通性。

### *连通与边点关系*

- 对于$n$个顶点的无向图，若其是连通图，则最少需要$n-1$条边，若其是非连通图，则最多有$C_{n-1}^{2}$条边（$n-1$个顶点构成一个完全图）。
- 对于$n$个顶点的有向图，若其是强连通图，则最少需要$n$条边来形成环路。
- 对于$n$个顶点的环，有$n$棵生成树。因为$n$个顶点的环的生成树的顶点为$n-1$，去掉任意一条边就能得到一棵生成树，环一共有$n$条边，所以可以去掉$n$条，得到$n$棵生成树。
- 对于$n$个顶点、$e$条边的无向图是一个森林，则一共有$n-e$棵树。设一共有$x$棵树，则只需要$x-1$条边就能将森林连接为一整棵树，所以边数+1 =顶点数（树的性质），即$e+(x-1)+1=n$，解得$x=n-e$。

### *连通概念关系*

- 有向完全图是强连通有向图。（完全就代表所有边都有双向弧，强连通代表所有边都有双向路径，条件更强）
- 生成树是原图的无环子图、极小连通子图且点集相同。（不是连通分量）

## 图的权

- 边的权：在一个图中，每条边都可以表上具有某种含义的数值，这就是该边的权值。

- 网络（网）：若图中的每条边都有权，这个带权图被称为网。
- 带权路径长度：取沿路径各边的权之和作为此路径的长度。

*无权图*

若$v[i][j]=0$，表示$v_{i+1}$到$v_{j+1}$是不连通的，若$v[i][j]=1$，表示$v_{i+1}$到$v_{j+1}$是连通的。

*网*

若$v[i][j]=\infty$，表示$v_{i+1}$到$v_{j+1}$是不连通的，若$v[i][j]=$某权值，表示$v_{i+1}$到$v_{j+1}$是连通的。

## 图的存储结构

### 邻接矩阵

*矩阵定义*

用一个一维数组保存顶点，用一个二维数组保存边，这个二维数组就是邻接矩阵。

使用一个长宽皆为$|v|$的二维矩阵$v$，从左上角到右上角，从左上角到左下角，分别标识表示$v_1, v_2 \cdots, v_n$。

对于无向图$v_{ij}=v_{ji}=1$，表示存在边$(v_i, v_j)$；对于有向图$v_{ij}=1$表示存在边$<v_i, v_j>$。

若$(v_i, v_j)$是$E(G)$中的边：对于无权图，矩阵$A[i][j]$存在就设为1，否则是0；对于有权图，矩阵$A[i][j]$存在就设为其权重$w_{ij}$，否则是0或∞；。

*矩阵性质*

度的性质：

- 对于无向图，第$i$个顶点的度=第$i$行或第$i$列的非零元素个数。
- 对于有向图，第$i$个顶点的出度=第$i$行的非零元素个数；第$i$个顶点的入度=第$i$列的非零元素个数；第$i$个顶点的度=第$i$行的非零元素个数+第$i$列的非零元素个数。

存储性质：

- 适用于存储稠密图。
- 对于无向图，因为没有方向，所以只有两点连接就是连通的，从而无向图的邻接矩阵都是主对角线对称的。因为对称，所以可以压缩存储。
- 对于无向图，图的边数等于上三角或下三角不包括主对角线的区域内的非零点的数量。

- 对于有向图，图的边数等于矩阵内所有非零点的数量。
- 空间复杂度是$O(|V|^2)$。
- 邻接矩阵存储图，很容易确定图中任意两个顶点之间是否有边相连，时间复杂度为$O(1)$。但是，要确定图中有多少条边，则必须按行、按列对每个元素进行检测，所花费的时间代价很大。
- 给定顶点找到其邻边要扫描一行，时间复杂度为$O(|V|)$。

设图$G$的邻接矩阵为$A$，矩阵元素为0或1，则$A^n$的元素$A^n[i][j]$表示由顶点$v_{i+1}$到顶点$v_{j+1}$的长度为$n$的路径的数目。

邻接矩阵的表示方式是唯一的。

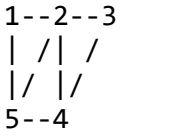
### 邻接表

对于稀疏图临界矩阵浪费空间，使用邻接表回更方便。

#### *表定义*

邻接表存储方式是顺序存储与链式存储的结合，存储方式和树的孩子表示法类似。

使用一个数组顺序保存图的每一个顶点，称为顶点表，使用链式存储让每一个顶点元素包含一个指向后一条边的指针，称为边表。




```
1→2→5^
2→1→5→3→4^
3→2→4^
4→2→5→3^
5→4→1→2^
```

#### *表性质*

度的性质：

- 对于无向图，每个顶点的边链表的顶点数就是该顶点的度。
- 对于有向图，每个顶点的边链表的顶点数就是该顶点的出度，而对于入度就只能遍历所有顶点的顶点链表。

存储性质：

- 对于无向图，因为同一条边两端的点会重复存储，所以空间复杂度为$O(|V|+2|E|)$，而对于有向图空间复杂度为$O(|V|+|E|)$。
- 给定顶点找到其邻边只需要读取其邻接表，时间复杂度为$O(1)$。
- 找到两个顶点是否存在邻边，需要在相应边表中查找另一个顶点。

邻接表的表示方式是不唯一的。

### 十字链表

十字链表置用于存储有向图。可以解决邻接矩阵空间复杂度高和邻接表计算入度入边不方便的问题。

十字链表定义了两种顶点：

- 顶点顶点：用于表示顶点，被一个数组包裹。
  - 数据域。
  - 该顶点作为弧头的第一条弧。
  - 该顶点作为弧尾的第一条弧。
- 弧顶点：被顶点顶点指向的顶点。
  - 弧尾顶点编号。
  - 弧头顶点编号。
  - 权值。
  - 弧头相同的下一条弧。
  - 弧尾相同的下一条弧。

空间复杂度为$O(|V|+|E|)$。

十字链表图表示是不唯一的。

### 邻接多重表

邻接多重表用于存储无向图，可以解决邻接矩阵空间复杂度高和邻接表删除插入顶点不方便的问题。

邻接多重表定义了两种顶点：

- 顶点顶点：用于表示顶点，被一个数组包裹。
  - 数据域。
  - 该顶点相连的第一条边。
- 边顶点：被顶点顶点指向的顶点。
  - 边一端编号$i$。
  - 边另一端编号$j$。
  - 权值。
  - 依附于$i$的下一条边。
  - 依附于$j$的下一条边。

空间复杂度为$O(|V|+|E|)$。

邻接多重表表示是不唯一的。

## 图的基本操作

### 图查找

#### *查找边*

使用邻接矩阵只用根据对应行列的元素是否为1或某值就可以了，如果是0或无穷，就代表没有该邻边。时间复杂度为$O(1)$。

而使用邻接矩阵需要从一端点出发遍历对应的顶点链表，如果能在链表中找到另一端点的索引，就代表有边。时间复杂度为$O(1)$到$O(|V|)$。

#### *查找点邻边*

对于无向图，邻接矩阵需要遍历对应顶点的那一行，所有数值为1或某数值的列就是对应的有边的另一个端点。时间复杂度为$O(|V|)$。

对于有向图，邻接矩阵需要遍历对应顶点的那一行得到出边以及那一列代表入边，所有数值为1或某数值的列就是对应的有边的另一个端点。时间复杂度为$O(|V|)$。

对于无向图，邻接表只用遍历对应顶点的顶点链表就可以。时间复杂度为$O(1)$到$O(|V|)$。

对于有向图，邻接表用遍历对应顶点的顶点链表得到出边，而对于入边需要遍历所有邻接表的边顶点。出边时间复杂度为$O(1)$到$O(|V|)$，入边时间复杂度为$O(|E|)$。

#### *查找头邻接点*

邻接矩阵只用扫描对应的行，找到顶点就可以了。时间复杂度为$O(1)$到$O(|V|)$。

对于无向图，邻接表只用找到顶点的边顶点的第一个顶点。时间复杂度为$O(1)$。

对于有向图，出边邻接表只用找到顶点的边顶点的第一个顶点。时间复杂度为$O(1)$。而对于入边需要遍历所有的顶点的第一个链表顶点。时间复杂度为$O(1)$到$O(|E|)$。

#### *查找下一个邻接点*

邻接矩阵只用扫描对应的行，找到顶点就可以了。时间复杂度为$O(1)$到$O(|V|)$。

邻接表只用找到当前顶点的下一个顶点。时间复杂度为$O(1)$。

### 图插入

邻接矩阵只用在最后增加一行一列。时间复杂度是$O(1)$。

邻接表只用在存储顶点的数组的末尾添加一个顶点，指针设置为$NULL$。时间复杂度是$O(1)$。

## 图删除

邻接矩阵的删除元素分为两种方式，如果是直接删除对应元素行与列上的所有元素并移动其他元素，那么时间复杂度就是$O(|V|^2)$，如果删除对应元素行与列上的所有元素但是不移动其他元素，而是将保存顶点数据的数组中对应顶点的数据变为$NULL$，则时间复杂度就是$O(|V|)$。

对于无向图，邻接表的删除需要删除该顶点并删除顶点后连接的所有顶点链表元素，时间复杂度为$O(1)$到$O(|V|)$。

对于有向图，邻接表的删除需要删除该顶点并删除顶点后连接的所有顶点链表元素且还要遍历所有的边并删除，删除出边时间复杂度为$O(1)$到$O(|V|)$，删除入边时间复杂度为$O(|E|)$。

## 图遍历

指从图某一顶点出发按照某种搜索方法沿着图中的边对图中所有顶点访问一次且仅访问一次。

树的遍历也可以看作一种特殊图的遍历。

分为广度优先$BFS$与深度优先$DFS$。

广度优先类似层序遍历，深度优先类似先序遍历。

广度优先是每一次遍历都要把所有的相邻顶点全部遍历到，而深度优先是每一次遍历只遍历最近的一个一直深入。

### *广度优先遍历*

实现条件：

- 找到一个与顶点相邻的所有顶点。
- 标记哪些顶点被访问过。
- 需要一个辅助队列保存顶点是否被访问的数据。

广度优先遍历过程：

1. 选择起始点并访问顶点$v$。
2. 访问$v$的所有未被访问的邻接点。
3. 依次从这些邻接点（在步骤二中访问的顶点）出发，访问它们的所有未被访问的邻接点；依此类推，直到图中所有访问过的顶点的邻接点都被访问。
4. 若图中尚未有顶点被访问，则另选一个未曾被访问的顶点作为起始点重复过程。

性质：

- 因为广度优先算法不需要回退，所以不是一个递归算法。
- 对于非带权图，使用$BFS$可以解决非带权图的单源最短路径问题，因为广度优先搜索按照距离有近到远。
- 邻接矩阵实现时的时间复杂度为$O(|V|^2)$，邻接表实现时的时间复杂度为$O(|V|+|E|)$；空间复杂度为$O(|V|)$。

相关概念：

- 广度优先生成树：根据广度优先遍历可以将所有第一次访问顶点时的路径组合生成一个广度优先生成树，若图顶点为$n$个，则生成树边一共有$n-1$条。因为保存图的数据结构若是不唯一，则其广度优先生成树也是不唯一的。若邻接矩阵存储则唯一，若邻接表存储则不唯一。
- 广度优先生成森林：若图是不连通的，那会生成连通分量个广度优先生成树，就构成了广度优先生成森林。

### *深度优先遍历*

实现条件：

- 是一个递归算法，所以需要一个工作栈。

深度优先遍历过程：

1. 访问顶点$v$。
2. 依次从$v$的未被访问的邻接点出发，对图进行深度优先遍历。
3. 直至图中和$v$有路径相通的顶点都被访问。
4. 若此时图中尚有顶点未被访问，则从一个未被访问的顶点出发，重新进行深度优先遍历，直到图中所有顶点均被访问过为止。

性质：

- 邻接表的深度优先序列会优先选择每个顶点的第一个相邻顶点，即顶点链中的第一个元素。
- 邻接矩阵方式唯一所以深度优先序列唯一，而邻接表方式不唯一，所以深度优先序列不唯一。
- 使用$DFS$算法递归地遍历一个无环有向图，并在退出递归时输出相应顶点，这样得到的顶点序列是逆拓扑有序。（因为栈的先进后出特性）
- 邻接矩阵实现时的时间复杂度为$O(|V|^2)$，邻接表实现时的时间复杂度为$O(|V|+|E|)$；空间复杂度为$O(|V|)$。

相关概念：

- 深度优先生成树：根据深度优先遍历可以将所有第一次访问顶点时的路径组合生成一个深度优先生成树，若图顶点为$n$个，则生成树边一共有$n-1$条。因为保存图的数据结构若是不唯一，则其深度优先生成树也是不唯一的。如

果无向图非连通，则一个顶点出发只能一次性遍历到该顶点所在连通分量的所有顶点。

- 深度优先生成森林：若图是不连通的，那会生成连通分量个深度优先生成树，就构成了深度优先生成森林。

图的广度优先生成树的高度小于等于深度优先生成树的高度。

*图遍历与图连通性*

- 若起始顶点到其他各顶点都有路径，那么只需调用一次深度优先或广度优先遍历函数。
- 对强连通图，从任意一顶点出发都只用调用一次深度优先或广度优先遍历函数。
- 遍历时函数调用层数等于该图的连通分量数。（因为存在不同的连通分量需要多次调用才能全部访问到）

## 图的应用

### 最小生成树

一个连通图的生成树包含图的所有顶点，并且只含尽可能少的边。对于生成树来说，若砍去它的一条边，则会使生成树变成非连通图；若给它增加一条边，则会形成图中的一条回路。

*最小生成树定义*

最小生成树$MST$也是最小代价树。已知生成树就是最小边的能到任意顶点的树，这种树只关心边数，所以有多个不同的生成树。

而最小生成树就是带权生成树的最小权值和的情况。

设$R$为图$G$的所有生成树的集合，若$T$为$R$中边的权值之和最小的生成树，则$T$称$G$的最小生成树。

- 最小生成树可能有多个，但是边的权值总是唯一且最小的。
- 最小生成树的边数=顶点数$-1$。减去一条则不连通，增加一条则会出现回路。
- 若一个连通图本身就是一棵树，则其最小生成树就是其本身。
- 只用连通图才有生成树，非连通图只有生成森林。
- 如果没有权值相同的边，则最小生成树是唯一的。

构建最小生成树的算法都利用的性质：假设$G=(V,E)$是一个带权连通无向图，$U$是顶点集$V$的一个非空子集。若$(u,v)$是一条具有最小权值的边，其中$u\in U$，$v\in V-U$，则必存在一棵包含边$(u,v)$的最小生成树。

获取最小生成树的$Prim$算法和$Kruskal$算法都是基于贪心算法的策略。每次都加入一条边逐渐生成一棵生成树：

```
function(G) {
  T=NULL;
  while T 未形成一棵生成树 {
    找到最小代价边(u,v)且加入 T 不会形成回路;
    T=T∪(u,v);
  }
}
```

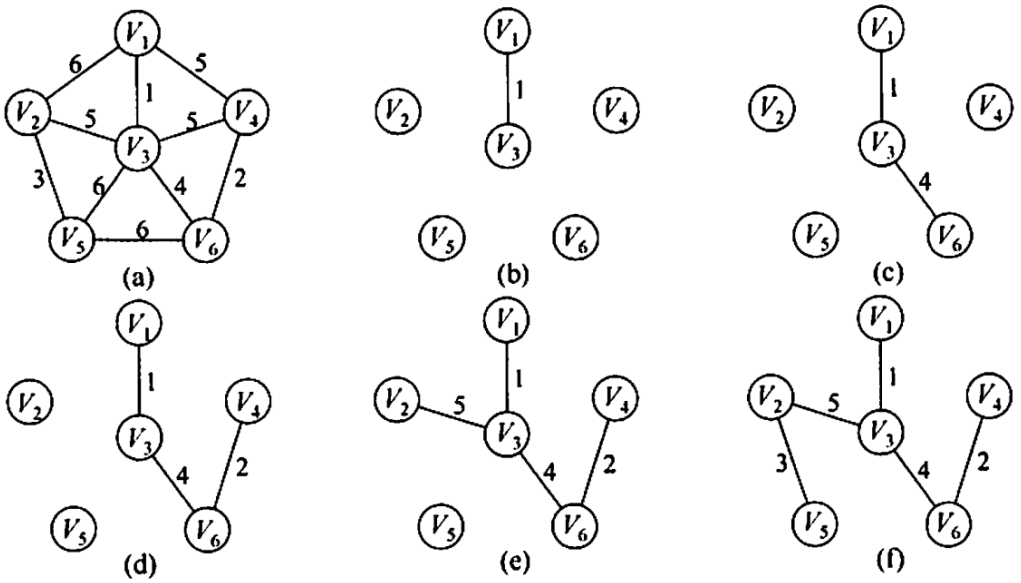
$MST$唯一性定理：$MST$没有使用无向网中相同权值的边，那么$MST$一定唯一。（连通图的任意一个环中所包含的边的权值均不相同）

### *Prim 算法*

非常类似找到图最短路径的迪杰斯特拉算法。

普里姆算法：从某个顶点开始构建生成树，每次将代价最小的新顶点纳入生成树，直到所有顶点都纳入为止。

- 假设$G=\{V,E\}$是连通图，其最小生成树$T=(U,E_T)$，$E_T$是最小生成树中边的集合。
- 初始化：向空树$T=(U,E_T)$中添加图$G=(V,E)$的任一顶点$u_0$，使$U=\{u_0\}$，$E_T=\varnothing$。
- 循环（重复下列操作直至$U=V$）：从图$G$中选择满足$\{(u,v)|u\in U,v\in V-U\}$且具有最小权值的边$(u,v)$，加入树$T$，置$U=U\cup\{v\}$，$E_T=E_T\cup\{(u,v)\}$。




*prim 算法*

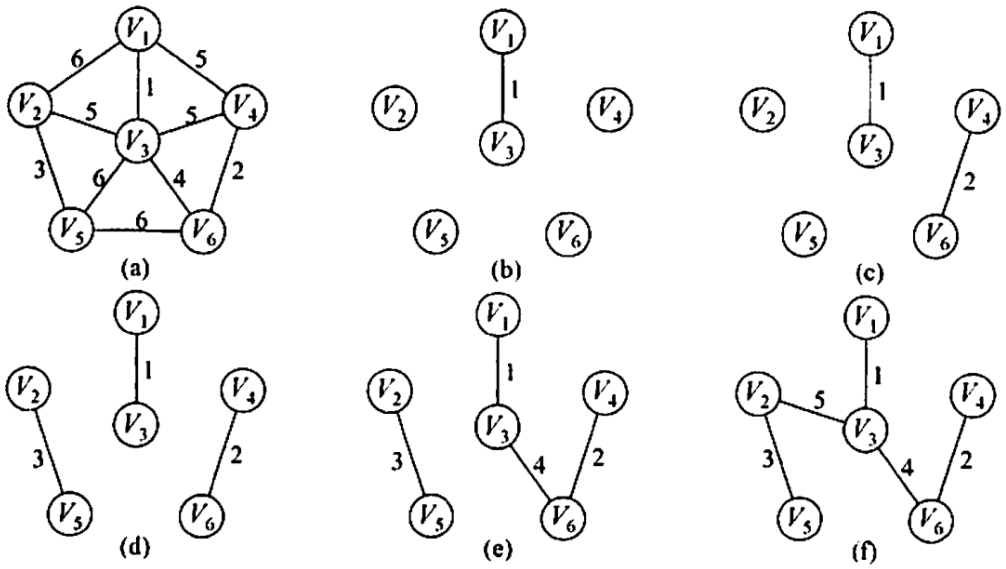
- 需要遍历$|V|$个顶点，每次要遍历其他所有顶点。
- 时间复杂度为$O(|V|^2)$。

- 适用于边稠密图。

### *Kruskal 算法*

克鲁斯卡尔算法：每次选择一条权值最小的边，使这条边的两头连通，若本就连通的就不选，直到所有的顶点都连通。

- 假设$G=(V,E)$是连通图，其最小生成树$T=(U,E_T)$。
- 初始化：$U=V,E_T=\varnothing$。即每个顶点构成一棵独立的树，$T$此时是一个仅含$|V|$个顶点的森林。
- 循环（重复下列操作直至$T$是一棵树）：按$G$的边的权值递增顺序依次从$E-E_T$中选择一条边，若这条边加入$T$后不构成回路，则将其加入$E_T$，否则舍弃，直到$E_T$中含有$n-1$条边。




*kruskal 算法*

- 使用堆来存放边（所以可以二分查找），所以每次旋转最小权值的边只需要$O(\log|E|)$的时间。
- 时间复杂度为$O(|E|\log_2|E|)$。
- 适用于边稀疏顶点多的图。

## 最短路径

- 单源最短路径：单个顶点到图的其他顶点的最短路径。
  - $BFS$算法（无权图）。
  - $Dijkstra$算法（带权图、无权图）。
- 每对顶点间最短路径：每对顶点之间的最短路径。
  - $Floyd$算法（带权图、无权图）。

最短路径一定是简单路径（不存在环）。但是无论有没有环的有向图与是否存在最短路径无关。

### *BFS 算法*

广度优先算法可以计算无权图的单源最短路径。

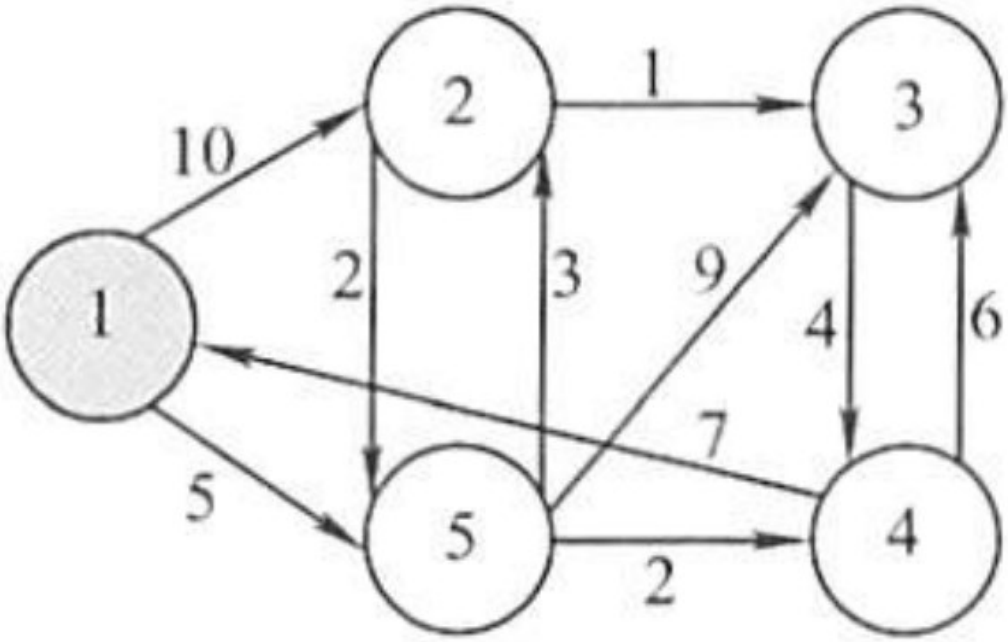
实际上无权图可以视为一种特殊的带权图，只是每条边的权值全部为1。

广度优先算法基本上就是对广度优先遍历的改进。定义两个数组，索引号就代表元素的序号，一个数组表示从起点开始到该点的最短路径长度，另一个数组表示从起点开始到该点的最短路径的上一个顶点的索引值。

### *Dijkstra 算法*

即迪杰斯特拉算法。用于计算单源最短路径（要计算从源到其他所有各顶点的最短路径长度）。

1. 从$v_0$开始，初始化三个数组：标记各顶点是否已找到最短路径$final$；最短路径长度$dist$；最短路径上的前驱$path$。从$v_0$开始，所以将$final[0] = true$，$dist[0] = 0$，$path[0] = -1$，然后将$v_0$直连的点的$dist$初始化为直连路径长度，对应的$path = 0$，但是不要将对应的$final = true$，因为还没有确定对应的直连路径就是最短路径。其他顶点的$dist = \infty$。
2. 遍历所有顶点，找到还没确定最短路径，且最短路径长度值最小的的一个顶点，这就确定了下一个最短路径的顶点，令其各顶点是否已找到最短路径的值为$true$。
3. 检查所有邻接这个顶点的其他顶点，若其点还没有找到最短路径，则更新最短路径长度值与最短路径上前驱的值。
4. 重复步骤二再次循环遍历所有顶点并找到没确定最短路径则最短路径长度最小的顶点。重复次数为$n - 1$次。




*dijkstra 算法*

| 顶点 | 第一轮 | 第二轮 | 第三轮 | 第四轮 |
|---|---|---|---|---|
| 2 权重 | 10 | 8 | 8√ | |
| 2 路径 | v1→v2 | v1→v5→v2 | v1→v5→v2 | |
| 3 权重 | ∞ | 14 | 13 | 9√ |
| 3 路径 | | v1→v5→v3 | v1→v5→v4→v3 | v1→v5→v2→v3 |
| 4 权重 | ∞ | 7√ | | |
| 4 路径 | | v1→v5→v4 | | |
| 5 权重 | 5√ | | | |
| 5 路径 | v1→v5 | | | |
| 集合 | {1,5} | {1,5,4} | {1,5,4,2} | {1,5,4,2,3} |

$Dijkstra$算法与$Prim$算法类似，都是优先与最短的路径结合。

时间复杂度为$O(|V|^2)$。

当权值中含有负权值的时候可能$Dijkstra$算法会失效。

### *Floyd 算法*

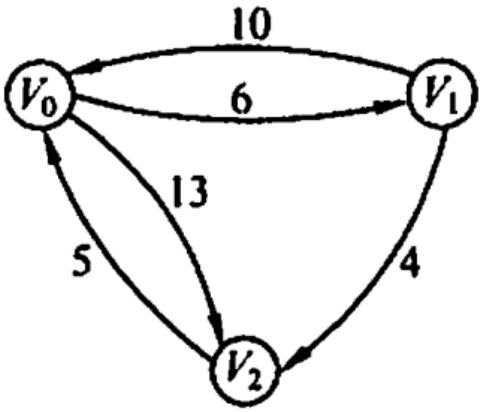
是一种动态规划算法，将问题的求解分为多个阶段。

对于$n$个顶点的图$G$，求任意一对顶点$v_i$到$v_j$之间的最短路径可分为如下阶段：

1. 初始化：不允许在其他顶点中转，求最短路径。
2. 若允许在$v_0$中转，求最短路径。
3. 若允许在$v_0$、$v_1$中转，求最短路径。
4. ...
5. 若允许在$v_0$、$v_1 \cdots v_{n-1}$中转，求最短路径。

算法需要遍历$n$次，每次遍历都需要查看$n \times n$的矩阵中是否有更优的中转点。

即判断若$A^{(k-1)}[i][j] > A^{(k-1)}[i][k] + A^{(k-1)}[k][j]$是否成立，若成立则$A^{(k)}[i][j] = A^{(k-1)}[i][k] + A^{(k-1)}[k][j]$，$path^{(k)}[i][j] = k$，否则$A^{(k)}$和$path^{(k)}$保持原样。其中$A^{(k)}$表示的是允许$v_0 \cdots v_k$个点中转后各顶点的最短路径长度，$path^{(k)}$表示允许$v_0 \cdots v_k$个点中转后两个点之间的中转点。




$$\begin{bmatrix} 0 & 6 & 13 \\ 10 & 0 & 4 \\ 5 & \infty & 0 \end{bmatrix}$$

*floyd 算法*

| 中转点 | -1 | | | 0 | | | 1 | | | 2 | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | v0 | v1 | v2 | v0 | v1 | v2 | v0 | v1 | v2 | v0 | v1 | v2 |
| v0 | 0 | 6 | 13 | 0 | 6 | 13 | 0 | 6 | 10 | 0 | 6 | 10 |
| v1 | 10 | 0 | 4 | 10 | 0 | 4 | 10 | 0 | 4 | 9 | 0 | 4 |
| v2 | 5 | ∞ | 0 | 5 | 11 | 0 | 5 | 11 | 0 | 5 | 11 | 0 |

$-1$代表是初始状态。

0代表用0进行中转，以0行和0列的数据都不用变，因为0行的数据表示$v0$到$v1$、$v2$的距离，0列的数据表示从$v1$、$v2$到$v0$的距离，这些不会以0进行中转。可以优化的只有12和21两个位置。同理对角线的元素都不用优化，固定为0因为是简单路径。12位置从$v0$中转是这个点对应行列的$v1v0 + v0v2 = 10 + 13 = 23 > 4$，所以不用优化。21位置原来为正无穷，所以可以优化为$v2v0 + v0v1 = 11$。

1代表用1进行中转，以1行和1列的数据都不用变，02位置若以$v1$中转则$v0v1 + v1v2 = 6 + 4 = 10 < 13$，所以优化为10。20位置若以$v1$中转则$v2v0 + v0v1 = 5 + 6 = 11 > 5$，不用优化。

2可以优化的位置为10、01，$v1v2+v2v0=4+5=9>10$，所以优化为10，$v0v2+v2v1=10+11=21>6$，不用优化。

时间复杂度为$O(|V|^3)$，空间复杂度为$O(|V|^2)$。

$Floyd$算法复杂度高，所以基本上都是四个顶点以下的图，能解决带负权值的问题，但是不能解决带有负权回路的图，即有负权值的边组成回路，这种图可能没有最短路径。

## 有向无环图

若一个有向图中不存在环，则是有向无环图，简称$DAG$图。

### *表达式应用*

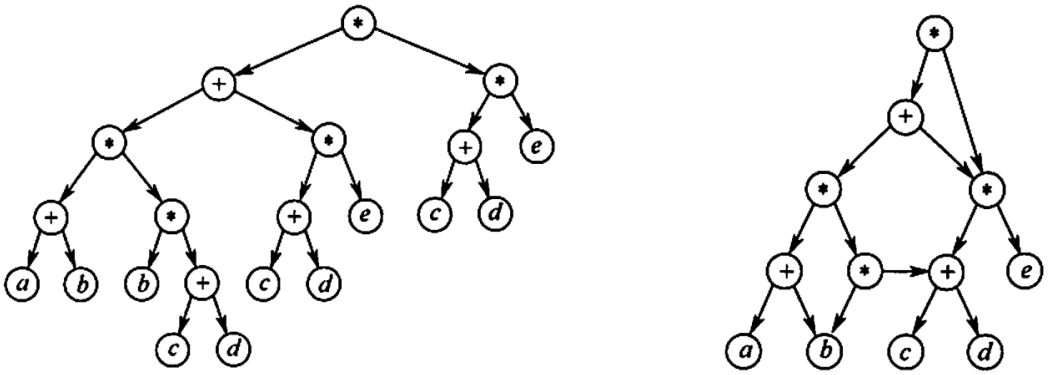
有向无环图可以运用到表达式的表达上，用树表示表达式，将操作数共同的顶点部分删除并将边合并到一起，这就形成了图，从而能精简表达式。

顶点中不可能出现重复的操作数。表达式树不唯一。

1. 把各个单个的操作数不重复的排成一排。
2. 标出各个运算符的生效顺序。
3. 按顺序加入运算符，并注意对运算符的优先级进行分层。
4. 当构建完成后从底向上逐层检查同层的运算符是否可以合并。

表达式：$((a+b)*(b*(c+d))+(c+d)"e)*((c+d)*e)$

（hq 批：这个表达式我也看不懂，建议结合上下文）



*表达式*

首先从第六层层有$c+d$与第五层有$c+d$重合，将第六层的$c+d$删掉，将第五层的乘号指向右边的加号上。第五层都有一个$b$，所以将加号和乘号都指向同一个$b$。然后往上面看还有相同的$(c+d)*e$，将其中一个删掉，连接到一起。

### *拓扑排序*

$AOV$网：用$DAG$图表示一个工程，顶点表示活动，有向边$< v_i, v_j >$表示活动$v_i$必须先于活动$v_j$进行。

拓扑排序：在图论中，由一个有向无环图的顶点组成的序列，当且仅当满足下列条件时，称为该图的一个拓扑排序：

1. 每个顶点出现且只出现一次。
2. 若顶点$A$在序列中排在顶点$B$的前面，则在图中不存在从顶点$B$到顶点$A$的路径。

或定义为：拓扑排序是对有向无环图的顶点的一种排序，它使得若存在一条从顶点$A$到顶点$B$的路径，则在排序中顶点$B$出现在顶点$A$的后面。每个$AOV$网都有一个或多个拓扑排序序列。

简单来说就是找到工程执行的先后顺序。

拓扑排序的实现：

1. 从$AOV$网中选择一个没有前驱的入度为0的顶点并输出。
2. 从网中删除该顶点和所有以它为起点的有向边。
3. 重复步骤一和二直到当前的$AOV$网为空或当前网中不存在无前驱的顶点（存在环路所以不能拓扑排序）为止。

逆拓扑排序的实现：

1. 从$AOV$网中选择一个没有后继的出度为0的顶点并输出。
2. 从网中删除该顶点和所有以它为起点的有向边。
3. 重复步骤一和二直到当前的$AOV$网为空或当前网中不存在无前驱的顶点为止。

若两个结点之间不存在祖先或子孙关系，则它们在拓扑序列中的关系是任意的（即前后关系任意)，因此使用栈和队列都可以保存结点，因为进栈或队列的都是入度为0的结点，此时入度为0的所有结点是没有关系的。

时间复杂度若使用邻接表为$O(|V| + |E|)$，若使用邻接矩阵则是$O(|V|^2)$)。

图和拓扑序列的关系：

- 如果有向图顶点不能排成一个拓扑序列，则有向图含有顶点大于1的强连通分量（即存在非自身环路）。
- 有向无环图的唯一拓扑序列不能唯一确定该图。

### *关键路径*

在带权有向图中，以顶点表示时间，以有向边表示活动，以边上的权值表示完成该活动的开销，称之为用边表示活动的网络，简称$AOE$网。

- 只有在某顶点所代表的事件发生后，从该顶点出发的各有向边所代表的活动才能开始。
- 只有在进入某顶点的各有向边所代表的活动都已经结束时，该顶点所代表的事件才能发生，活动也可以并行进行。
- 只有一个入度为0的顶点，即开始顶点（源点），表示整个工程的开始。
- 只有一个出度为0的顶点，称为结束顶点（汇点），表示整个工程的结束。
- 从源点到汇点的有向路径可能有多条，所有路径中，具有最大路径长度的路径称为关键路径（即决定完成整个工程所需的最小时间），而关键路径上的活动称为关键活动。
- 事件的最早发生时间：决定了所有从该事件开始的活动能够开工的最早时间。一个事件的最早发生时间与以该事件为始的弧的活动的最早开始时间相同。
- 活动的最早开始时间：指该活动弧的起点所表示的事件最早发生时间。
- 事件的最迟发生时间：在不推迟整个工程完成的前提下，该事件最迟必须发生的时间。一个事件的最迟发生时间等于min{以该事件为尾的弧的活动的最迟开始时间，最迟结束时间与该活动的持续时间的差}。
- 活动的最迟开始时间：指该活动弧的终点所表示的事件的最迟发生时间与该活动所需时间之差。
- 活动的时间余量：在不增加完成整个工程所需总时间的情况下，活动可以拖延的时间。

求关键路径的步骤：

1. 求所有事件的最早发生时间$ve$。根据拓扑排序序列，依次按照所有路径的最大值求出各个顶点的最早发生时间。
2. 求所有事件的最迟发生时间$vl$。根据逆拓扑排序序列，回退依次将每个顶点按第一步计算的整个工程的时间减去本顶点需要处理的时间，得到每个活动都最晚应该发生的时间，交叉的顶点取最小值。
3. 求所有活动的最早发生时间$e$。根据第二步可以得到每个活动发生的最早时间。
4. 求所有活动的最迟发生时间$l$。根据$vl$求每个顶点不影响整个工程的情况下最晚可以什么时候开始。
5. 求所有活动的时间余量$d$。将$l-e$得到$d$，余量为0的活动就是关键活动，表示如果该活动拖延就会影响整个工程的进度。

关键活动在关键路径上。

- 关键活动时间增加，整个工程工期延长。

- 关键活动时间减少，整个工程工期缩短。
- 关键活动时间减少，可能变为非关键活动。
- 若有多条关键路径，则必须提高所有关键路径关键活动才能缩短工期。
- 键路径是从源点到汇点路径长度量长的路径。

*环判定*

无向图回路的判断：

1. 在图的邻接表表示中，首先统计每个顶点的度，然后重复寻找一个度为1的顶点，将度为1和0的顶点从图中删除，并将与该顶点相关联的顶点的度减1，然后继续反复寻找度为1的。如果最后存在点没有被删除，即在寻找过程中若出现若干顶点的度都为2，则这些顶点组成了一个回路；否则，图中不存在回路。
2. 利用深度优先搜索$DFS$，在搜索过程中判断是否会出现后向边（$DFS$中，连接顶点$u$到它的某一祖先顶点$v$的边），即在$DFS$对顶点进行着色过程中，若出现所指向的顶点已经着色，则此顶点是一个已经遍历过的顶点（祖先），出现了后向边，则图中有回路。
3. 利用$BFS$，在遍历过程中，为每个节点标记一个深度$deep$，如果存在某个节点为$v$，除了其父节点$u$外，还存在与$v$相邻的节点$w$使得$deep[v] <= deep[w]$（可以通过相邻点上升返祖）的，那么该图一定存在回路。
4. 用$BFS$或$DFS$遍历，最后判断对于每一个连通分量当中，如果边数$m >=$节点个数$n$成立，那么改图一定存在回路。因此在$DFS$或$BFS$中，我们可以统计每一个连通分量的顶点数目$n$和边数$m$两个直值，如果$m >= n$则返回假，直到访问完所有的节点才返回真。

有向图回路的判断：

1. 与无向图类似，若在$DFS$中出现后向边，即存在某一顶点被第二次访问到，则有回路出现。
2. 同样，利用拓扑排序的思想，通过这一步骤来执行拓扑排序，即重复寻找一个入度为0的顶点，将该顶点从图中删除，并将该顶点及其所有的出边从图中删除（即与该点相应的顶点的入度减1），最终若途中全为入度为1的点，则这些点至少组成一个回路。

# 查找

## 基本概念

- 查找：在数据集合中寻找满足某种条件的数据元素的过程。
- 查找表（查找结构）：用于查找的数据集合，由同一类型的数据元素或记录组成。

- 关键字：数据元素中唯一标识该元素的某个数据项的值，使用基于关键字的查找，查找结果应该唯一。
- 静态查找表：只查找符合条件的数据元素。（顺序查找、折半查找、散列查找）
- 动态查找表：出来要查找，还要进行删除或插入，除了查找速度还要考虑插入删除操作是否方便。（二叉查找树查找、散列查找）
- 查找长度：查找运算中，需要对比关键字的次数。
- 平均查找长度$ASL$：所有查找过程中进行关键字比较次数的平均值。$ASL = \sum_{i=1}^{n} P_i\, C_i$。其中$P_i$表示查找第$i$个元素的概率，$C_i$表示查找第$i$个元素的查找长度。

## 线性表查找

### 顺序查找

又称为线性查找，常用于线性表，从头到尾逐个查找。

#### *一般查找*

在算法实现时一般将线性表的0号索引的元素值设为查找的值，从表最后面开始向前查找，当没有找到时就直接从0返回。这个数据称为哨兵，可以避免不必要的判断线性表长是否越界语句从而提高程序效率。

$ASL$查找成功为$\frac{1+2+3+\cdots+n}{n} = \frac{n+1}{2}$，$ASL$查找失败为$n+1$，时间复杂度为$O(n)$。

#### *顺序表*

可以让数据集变为有序的，这样对比数据大小也可以知道是否还需要遍历从而减少查找时间。这样顺序结构从逻辑上就变成了一个二叉树结构（一般是顺序的），左子树代表小于（失败结点），右子树代表大于（需要继续向右查找），所有数据结点挂在右子树上。

对于顺序查找，无论顺序表还是乱序表，查找成功的时间都是相同的。$ASL$也是相同的。

若有$n$个结点，则有$n+1$个查找失败结点。

查找情况还要比普通乱序查找加上一个大于最大值的情况。

虽然有$n+1$个失败结点，但是失败结点都是不存在的，如果结点失败只会在一个不存在的区间上，所以查找平均概率是$\frac{1}{n+1}$而不是$\frac{1}{2n+1}$。同理，查找成功查找失败都会在那个区间停下，一旦判断所在区域不存在就会退出，所以查找失败的次数跟最坏查找成功次数一样都是$n$查完所有。

$ASL$查找失败为$\frac{1+2+3+\cdots+n+n}{n+1}=\frac{n}{2}+\frac{n}{n+1}$。成功结点的查找长度等于自身所在层数，失败结点的查找长度等于其父结点所在层数。

注意，有序线性表的顺序查找和后面的折半查找的思想是不一样的（有序顺序查找是从两端开始向右比较，折半查找从中间开始不断取中间值比较）。且有序线性表的顺序查找中的线性表可以是链式存储结构。

### *查找概率不等*

当数据元素查找概率不等时，可以将查找概率更大的元素放在靠前的位置，以减少大概率元素被遍历的时间。

$ASL$查找成功为$\sum_{i=1}^{n}P_i\,i$，$P_i$为第$i$个元素出现概率。

但是此时数据是乱序的，所以查找失败的时间复杂度与没有优化的是一样的。

## 折半查找

也称为二分查找，只适用于有序的顺序表，链表无法适用，因为链表很难折半找到元素。

### *折半查找的结构*

定义三个指针，$low$指向查找范围的最小值，$high$指向查找范围的最大值，$mid$指向查找范围的中间值，$mid=\lfloor(low+high)\div 2\rfloor$。（也可以向上取整，过程会有所不同）

### *折半查找的过程*

1. 定义左边界$low$，默认为0，右边界$high$，默认为$length-1$，循环执行折半查找（2，3 两步）。
2. 计算出$mid=\lfloor(low+high)\div 2\rfloor$。
3. 判断中间索引值$data[mid]$是否与搜索值$target$相等。
   - 若$data[mid]=target$，返回中间索引。
   - 若$data[mid]<target$，则将$high=mid-1$。
   - 若$data[mid]>target$，则将$low=mid+1$的值。
4. 当查找最后$low>high$则查找失败。

$ASL$查找成功为$\frac{1+2+3+\cdots+n}{n}=\frac{n+1}{2}$，$ASL$查找失败为$n+1$，时间复杂度为$O(n)$。

### *折半公式优化*

在对$mid$进行取值时，如果数据量太大，查找到右侧时计算$mid$进行两数相加$low+high$可能会数值溢出。那么如何解决？

- 变幻公式：$(low + high) \div 2 \rightarrow low \div 2 + high \div 2$或$\rightarrow low - (low \div 2 - high \div 2) \rightarrow low + (high - low) \div 2$。
- 无符号右移运算：$mid = (low + hight) >>> 1$。直接将除以2变为右移运算，速度更快，且舍去了小数位不需要进行取整运算。

### *折半查找判定树*

折半查找的过程可用二叉树来描述，称为判定树。树中每个圆形结点表示一个记录，结点中的值为该记录的关键字值；树中最下面的叶结点都是方形的，它表示查找不成功的情况。从判定树可以看出，查找成功时的查找长度为从根结点到目的结点的路径上的结点数，而查找不成功时的查找长度为从根结点到对应失败结点的父结点的路径上的结点数。

构建判定树方式：

- 若当前$low$和$high$有奇数个元素，则$mid$分割后左右两部分元素个数相等。
- 若当前$low$和$high$有奇数个元素，则$mid$分割后左部分元素个数小于右部分一个。
- 数值结点为圆形，末端结点后再加一层方形的叶子结点，表示查找失败。

判定树性质：

- 折半查找判定树一定是一个平衡二叉树。只有最下面一层不满，元素个数为$n$时树高与完全二叉树相等$h = \lceil \log_2(n+1) \rceil$。
- 根据折半查找判定树可以计算对应的$ASL$：查找成功的$ASL$=($\sum_{i=1}^{n}$第$i$层的成功结点数$\times i$)$\div$成功结点总数，查找失败的$ASL$=($\sum_{i=1}^{n}$第$i$层的失败结点数$\times i$)$\div$失败结点总数。
- 折半查找判定树也是一个二叉查找树，失败结点$= n + 1$（成功结点的空链域结点数）。
- 折半查找判定树的中序序列应该是一个有序序列。

$ASL$查找成功查找失败都一定小于折半查找树的树高，时间复杂度为$O(\log_2 n)$。

查找的折半查找判定树和排序的二叉查找树都是同样的二叉逻辑结构，但是不同的是折半查找判定树是已知完整序列，所以总是从中间开始，时间性能为固定的$O(\log_2 n)$，而二叉查找树的构造是根据输入来的，如果输入的序列正好是从中间切分的，则时间性能为最好的$O(\log_2 n)$，如果输入的序列恰好有序，则为单枝树，时间性能为最坏的$O(n)$。

### 分块查找

分块查找又称为索引顺序查找，需要对数据进行一定的排序，不一定全部是顺序的，但是要求在一个区间内是满足一定条件的，即块内无序，块间有序。即$n$块内的元素全部小于$n+1$块内的任意元素。

将查找表分割为若干子块，其中分割的块数和每块里的数据个数都是不定的。

### *分块查找结构*

除了保存数据的顺序表外还需要定义一个索引表，保存每个分块的**最大关键字**、每块的存储空间，第一个元素的地址。

很明显这种定义方式定义了两个顺序结构，并且如果插入删除时需要大量移动元素，所以可以采用链表的形式。

定义一种最大元素结点，包含数据、指向后继最大元素结点的指针、指向分块内元素的指针；定义一种块内元素结点，包含数据、指向后继分块内元素的指针。但是这时候就无法折半查找，只能顺序查找。

所以总的来说分块查找还是一种静态查找，动态插入删除的效率较低。

### *分块查找过程*

在查找时先根据关键字遍历索引表，然后找到索引表的分块（可以顺序也可以折半），再到存储数据的顺序表的索引区间中查找。

若适用折半查找查找索引表的分块，索引表中若不存在目标关键字，则折半查找索引表最终会停在$low > high$，要在$low$所指向分块中查找。

### *分块查找的效率*

$ASL$查找成功失败的情况都十分复杂，所以一般不会考。

假设长度为$n$的查找表被均匀分为$b$块，每块$s$个元素，假设索引查找和块内查找的平均查找长度$ASL$分别为$L_I$和$L_S$，则分块查找的平均查找长度为$ASL = L_I + L_S$。

使用顺序查找索引表，则$L_I = \frac{(1+2+\cdots+b)}{b} = \frac{b+1}{2}$，$L_S = \frac{(1+2+\cdots+s)}{s} = \frac{s+1}{2}$。所以$ASL = \frac{b+1}{2} + \frac{s+1}{2} = \frac{s^2+2s+n}{2s}$，当$s = \sqrt{n}$时，$ASL_{min} = \sqrt{n} + 1$。

使用折半查找索引表，则$L_I = \lceil \log_2(b+1) \rceil$，$L_S = \frac{(1+2+\cdots+s)}{s} = \frac{s+1}{2}$。所以$ASL = \lceil \log_2(b+1) \rceil + \frac{s+1}{2}$。

## 二叉查找

### 二叉查找树

即$BST$，是一种用于排序的二叉树。

### *二叉查找树定义*

二叉查找树也是二叉排序树。左子树上所有结点的关键字均小于根结点的关键字；右子树上所有结点的关键字均大于根结点的关键字；左右子树又各是一棵二叉查找树。

中序遍历二叉查找树会得到一个递增的有序序列。

### *二叉查找树查找*

1. 若树非空，目标值与根结点的值比较。
2. 若相等则查找成功，返回结点指针。
3. 若小于根结点，则在左子树上查找，否则在右子树上查找。
4. 遍历结束后仍没有找到则返回$NULL$。

遍历查找的时间复杂度是$O(\log_2 n)$，则递归查找的时间复杂度是$O(\lceil \log_2(n+1) \rceil)$，其中$\lceil \log_2(n+1) \rceil$代表二叉树的高度。

查找成功的平均查找长度$ASL$，二叉树的平均查找长度为$O(\log_2 n)$，最坏情况是每个结点只有一个分支，平均查找长度为$O(n)$。

### *二叉查找树插入*

- 若原二叉查找树为空，就直接插入结点。
- 否则，若关键字小于根结点值，插入左结点树。
- 若关键字大于根结点值，插入右结点树。

### *二叉查找树删除*

- 搜索到对应值的目标结点。
- 若被删除结点$p$是叶子结点，则直接删除，不会破坏二叉查找树的结构。
- 若被删除结点只有一棵左子树或右子树，则让该结点的子树称为该结点父结点的子树，来代替其的位置。
- 若被删除结点有左子树和右子树，则让其结点的直接后继（中序排序该结点后一个结点，其右子树的最左下角结点，不一定是叶子结点）或直接前驱（中序排序该结点前一个结点，其左子树的最右下角结点，不一定是叶子结点）替代该结点，并从二叉查找树中删除该的结点直接后继、直接前驱，这就变成了第一种或第二种情况。

二叉查找树删除或插入时得到的二叉查找树往往与原来的不同。

### *二叉查找树查找效率*

二叉查找树的查找效率主要取决于树的高度，若是平衡二叉树，则平均查找长度为$O(\log_2 n)$，若最坏情况下只有一个单枝树，则平均查找长度为$O(n)$。

若按顺序输入关键字则会得到单枝树。

查找过程来看，二叉查找树和二分查找类似。但是二分查找的判定树唯一，而二叉查找树的查找不唯一，相关关键字插入顺序会极大影响到查找效率。

从维护来看，二叉查找树插入删除操作只需要移动指针，时间代价为$O(\log_2 n)$，而二分查找的对象是有序顺序表，代价是$O(n)$。

所以静态查找时使用顺序表进行二分查找，而动态查找时使用二叉查找树。

## 平衡二叉树

为了限制判定树高增长过快，降低二叉查找树的性能，规定插入时要保证平衡。

即$AVL$树，树上任意一结点的左子树和右子树的高度之差不超过1。

树高即树的深度。

结点的平衡因子=左子树高-右子树高。

即平衡二叉树的平衡因子只可能为$-1,0,1$。

在插入一个结点时，查找路径上的所有结点都可能收到影响。

从插入点往回（从下往上）找到第一个不平衡的结点，调整以该结点为根的子树。每次调整的对象都是最小不平衡树。

### *平衡二叉树结点*

$h$为平衡二叉树高度，$n_h$为构造此高度的平衡二叉树所需的最少结点数。

- 平衡二叉树最少结点数（所有非叶结点的平衡因子均为1的）的递推公式为$n_0 = 0$，$n_1 = 1$，$n_2 = 2$，$n_h = 1 + n_{h-1} + n_{h-2}$。此时所有非叶结点的平衡因子均为1。

假设$T$为高度为$h$的平衡二叉树，其需要最少的结点数目为$F(h)$。

又假设$TL$，$TR$为$T$的左右子树，因此$TL$，$TR$也为平衡二叉树。

假设$FL$、$FR$为$TL$，$TR$的最少结点数，则，$F(h) = FL + FR + 1$。那么$FL$、$FR$到底等于多少呢？

由于$TL$，$TR$与$T$一样是平衡二叉树，又由于我们知道$T$的最少结点数是$F(h)$，其中$h$为$T$的高度，因此如果我们知道$TL$，$TR$的高度就可以知道$FL$、$FR$的值了。

由平衡二叉树的定义可以知道，$TL$和$TR$的高度要么相同，要么相差1，而当$TL$与$TR$高度相同（即都等于$h-1$）时，我们算出来的$F(h)$并不能保证最小，两边都是$h-1$明显比只有一边$h-2$的结点数更多。

因此只有当$TL$与$TR$高度相差一，即一个高度为$h-1$，一个高度为$h-2$时，计算出来的$F(h)$才能最小。

此时我们假设$TL$比$TR$高度要高1，即$TL$高度为$h-1$，$TR$高度为$h-2$，则有$F1=F(h-1)$，$F2=F(h-2)$。

因此得到结论：$F(h)=F(h-1)+F(h-2)+1$。

- 平衡二叉树最多结点数$2^n-1$。即该二叉树为满二叉树。

### *平衡二叉树插入*

最重要的是根据二叉查找树的大小关系算出从大到小的序列，然后把最中间的作为新根，向两侧作为左右子树。

即先找到插入路径上离插入结点最近的平衡因子的绝对值大于1的结点$A$，再对以$A$为根的子树，在保持二叉查找树特性的前提下调整各结点位置。

每次调整的都是最小不平衡子树。

### *平衡二叉树删除*

与插入操作类似，都是需要从下往上进行调整。

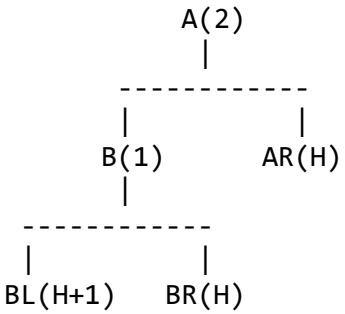
不同的是插入操作只对子树进行调整，而删除操作可能要对整个树进行调整。

在任意一棵非空二叉查找树$T_1$中，删除某结点$v$之后形成二叉查找树$T_2$，再将$v$插入$T_2$，形成二叉查找树$T_3$。

- 若$v$是$T_1$的叶结点，则$T_1$与$T_3$相同。
- 若$v$不是$T_1$的叶结点，则$T_1$与$T_3$不同（如果是一定不同则是错误的）。

### *右单旋转*

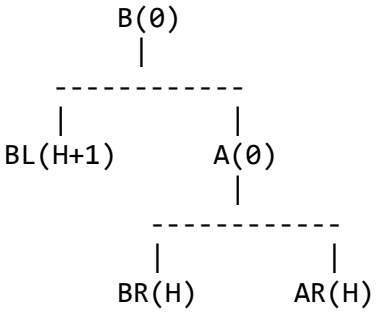
从结点的左孩子的左子树中插入导致不平衡：




$$BL<B<BR<A<AR$$

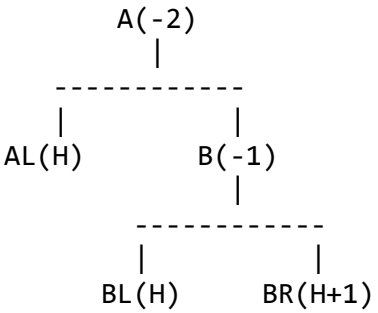
由于在结点$A$的左孩子$B$的的左子树$BL$上插入了新结点，$A$的平衡因子由1变成了2，导致以$A$为根的子树失去了平衡，需要进行一次向右的旋转操作。

将$A$的左孩子$B$向右上旋转代替$A$成为根结点，将$A$结点向右下选择为成$B$的右子树的根结点，而$B$的原右子树则作为$A$结点的左子树。




*左单旋转*

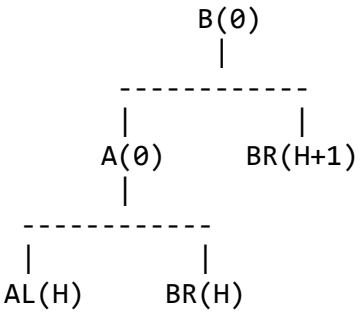
从结点的右孩子的右子树中插入导致不平衡：




$$AL < A < BL < B < BR$$

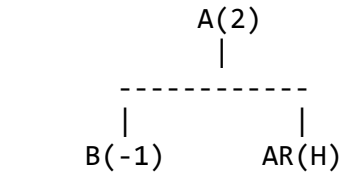
由于在结点$A$的右孩子$R$的右子树$R$上插入了新结点，$A$的平衡因子由$-1$减至$-2$，导致以$A$为根的子树失去平衡，需要一-次向左的旋转操作。

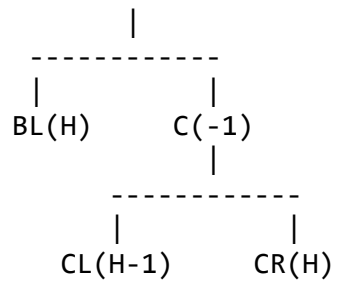
将$A$的右孩子$B$向左上旋转代替$A$成为根结点，将$A$结点向左下旋转成为$B$的左子树的根结点，而$B$的原左子树则作为$A$结点的右子树。

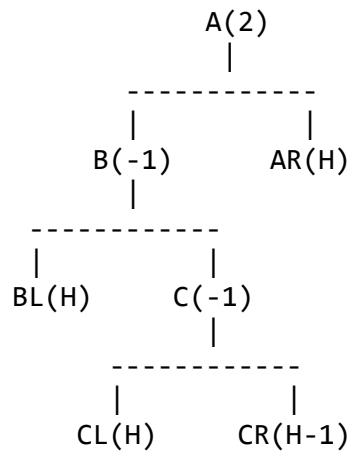



*先左后右双旋转*

从结点的左孩子的右子树中插入导致不平衡：



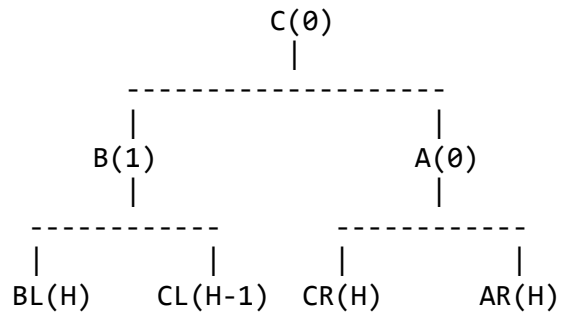


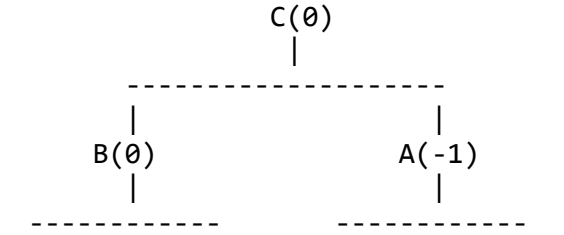




将$BR$拆分为$C$和$CL$、$CR$，假设插入的是$CR$部分，插入$CL$也同理。

$$BL < B < CL < C < CR < A < AR$$

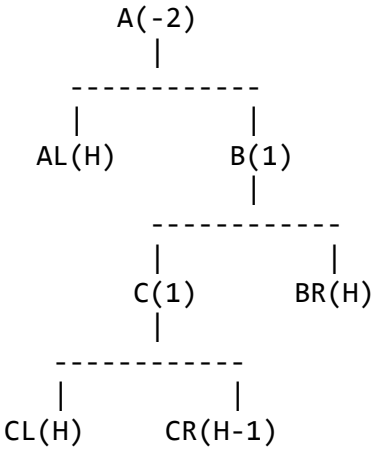
由于在$A$的左孩子$L$的右子树$R$上插入新结点，$A$的平衡因子由1增至2，导致以$A$为根的子树失去平衡，需要进行两次旋转操作，先左旋转后右旋转先将$A$结点的左孩子$B$的右子树的根结点$C$向左上旋转提升到$B$结点的位置，然后再把该$C$结点向右上旋转提升到$A$结点的位置。

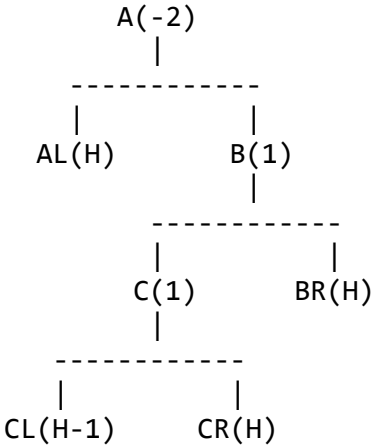



*先右后左双旋转*

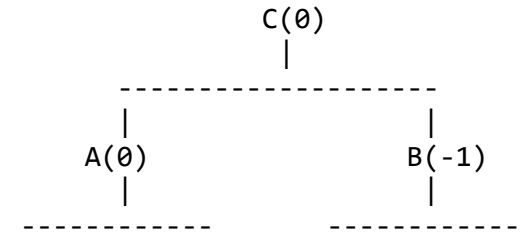
从结点的右孩子的左子树中插入导致不平衡：

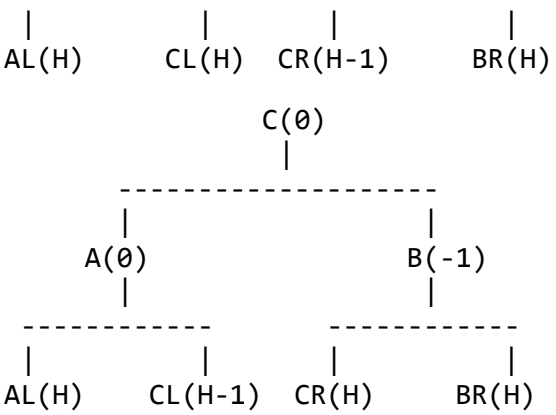



$$AL < A < CL < C < CR < B < BR$$

由于在$A$的右孩子$R$的左子树$L$. 上插入新结点，$A$的平衡因子由$-1$减至$-2$，导致以$A$为根的子树失去平衡，需要进行两次旋转操作，先右旋转后左旋转。先将$A$结点的右孩子$B$的左子树的根结点$C$向右上旋转提升到$B$结点的位置，然后再把该$C$结点向左上旋转提升到$A$结点的位置。






*平衡二叉树查找*

含有$n$个结点的平衡二叉树最大深度为$O(\log_2 n)$，平均查找长度为$O(\log_2 n)$。

## 红黑树

*红黑树定义*

为了维持二叉平衡树，需要反复对树整体进行插入合删除，代价巨大，所以引入为弱化版相对平衡的二叉查找树——红黑树：

1. 每个结点或是红色，或是黑色的。
2. 根结点是黑色的。
3. 叶结点（$n+1$个虚构的外部结点、$NULL$结点）都是黑色的，保证红黑树的内部结点左右孩子均非空。
4. 不存在两个相邻的红结点（即红结点的父结点和孩子结点均是黑色的）。
5. 对每个结点，从该结点到任一叶结点的简单路径上，所含黑结点的数量相同。

所以定义某结点出发到达一个叶结点的任一简单路径上的黑结点总数（**不含**该目的结点）称为该结点的黑高（记为$bh$），根结点的黑高就是红黑树的黑高。

*红黑树性质*

从根到叶结点的最长路径不大于最短路径的两倍。

- 由性质⑤，当从根到任一叶结点的简单路径最短时，这条路径必然全由黑结点构成（即第二层的结点）。
- 由性质④，当某条路径最长时，这条路径必然是由黑结点和红结点相间构成的，此时红结点和黑结点的数量相同（非第二层的其他所有结点）。

有$n$个内部结点的红黑树的高度$h \le 2\log_2(n+1)$。

- 若红黑树的总高度为$h$，则根结点黑高$\ge \frac{h}{2}$，所以内部结点$n \ge 2^{\frac{h}{2}-1}$（假设没有红结点），所以$h \le 2\log2(n+1)$。

红黑树查找、插入、删除的时间复杂度都是$O(\log_2 n)$。

- 插入和删除：由于红黑树的每次操作平均要旋转一次和变换颜色，而普通二叉查找树如果平衡因子在指定范围内不会旋转（如果要旋转则可能旋转多次），所以它比普通的二叉查找树效率要低一点，不过时间复杂度仍然是$O(\log_2 n)$。
- 普通查询：没有使用到红黑树的性质，所以红黑树和二叉查找树的效率相同。对于平衡树而言，平衡树的效率更高。
- 插入数据有序查询：红黑树的查询效率就比二叉搜索树要高了，因为此时二叉搜索树不是平衡树，它的时间复杂度$O(n)$。

### *红黑树插入概述*

假定当前结点为$x$，其父结点为$p$，叔父结点（父结点的兄弟结点）为$u$，祖父结点为$g$。

红黑树的插入过程和二叉查找树的插入过程基本类似，从右至左，右上至下。不同之处在于，红黑树中插入新结点后需要进行调整（主要通过重新着色或旋转操作进行），以满足红黑树的性质。

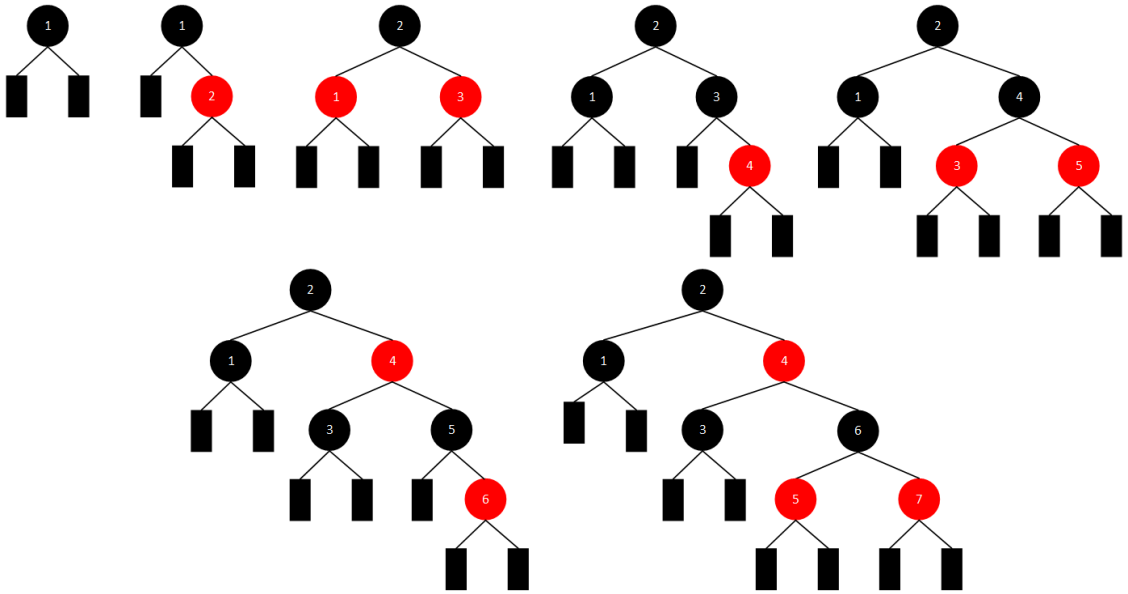
- 插入红黑树中的结点$z$初始着为红色。

假设新插入的结点初始着为黑色，那么这个结点质在的路径比其他路径多出一个黑结点（几乎每次插入都破坏性质⑤），调整起来也比较麻烦。如果插入的结点是红色的，此时所有路径上的黑结点数量不变，仅在出现连续两个红结点时才需要调整，而且这种调整也比较简单。

| 父结点 | 叔结点 | 插入类型 | 操作 |
|---|---|---|---|
| 黑 | - | - | 无需操作 |
| 红 | 黑 | 左左 | 右旋，变色 |
| 红 | 黑 | 右右 | 左旋，变色 |
| 红 | 黑 | 左右 | 左旋，右旋，变色 |
| 红 | 黑 | 右左 | 右旋，左旋，变色 |
| 红 | 红 | - | 父叔变红，祖父变黑，祖父变为当前结点，递归调整 |

插入全为红，是根则变黑；父黑无变化，父红看叔叔；叔红只变色，叔父爷全变，爷爷变儿子，重新来一遍；叔黑看平衡，看谁最后转，谁转谁变色。

以1 ~ 7的序列构建红黑树：




*红黑树插入*

虽然有六种情况，但是由于两种是镜像的，所以归纳为四种。

### *父结点黑*

默认插入结点为红色，且父结点为黑，所以此时插入后不影响黑高和平衡，从而不需要调整。

插入2，只有一个父结点为根结点，没有叔结点，直接插入到1的右子树位置，没有其他变化。

### *父结点红叔结点黑同侧插入*

即插入类型为左左或右右，即在其祖父结点的左子树的左侧插入元素或其祖父结点的右子树的右侧插入元素。

由于插入结点为红色，父结点也为红色，不能出现连续的红色，所以就要变色，此时就应该先旋转。

由于是同侧插入，所以应该往另一侧旋转，旋转方式与二叉查找树一样，旋转完成后重新涂色。

插入3，3插入2的右子树位置，其父结点2为红色，而叔结点为1的右$NULL$结点为黑色，所以需要旋转，由于插入的是2的右侧，而2是其父结点的右子树，所以是同侧插入，需要反方向向左旋转，2结点上升变为根结点，2的左侧$NULL$结点给1结点的右侧，1下沉，3同样上升，完成旋转。然后涂色，根结点2为黑色，$NULL$为黑色，所以中间的1和3都涂成红色。

### *父结点红叔结点黑异侧插入*

即插入类型为左右或右左，即在其祖父结点的左子树的右侧插入元素或其祖父结点的右子树的左侧插入元素。

此时红黑树末端会不平衡，此时就需要向插入的同侧方向旋转一下，从而变成父结点红树结点黑同侧插入的情况，再按照这个类型的处理方式进行处理。

插入5，注意此时图中是按同侧插入，即5是插入到4这个右子树的右侧的，而现在我们将5插入到4的左子树位置。5的父结点4是红色的，其叔结点3的左$NULL$结点是黑色的，而5插入的位置是4所在右子树的结点左侧，所以符合异侧插入的处理方式。首先对4和5这个子树进行右旋，将5右侧的$NULL$给4的左子树位置，5上升变成3的子结点，4下沉变成5的子结点。此时就是同侧插入插入的情况。再进行左旋，5的左$NULL$结点给当前父结点3的右子树位置，5上升变为父结点，变成2的子结点，3、4下沉变为5的子结点。此时红黑树形态一样，但是2的子结点从1和4变成了1和5。

### *父结点红叔结点红*

由于插入结点为红色，父结点也为红色，不能出现连续的红色，所以就要变色。

这种情况是不需要进行旋转的。

由于父结点和叔结点都是红色，不能连续红色，所以将父结点和叔结点都涂为黑色；由于父结点和叔结点都是红色，所以其祖父结点一定是黑的，涂为黑色。此时将当前结点移动到祖父结点上，查看先祖结点的涂色是否正确，递归这个逻辑直到根结点，根结点一定是黑色的。

插入4，4插入3的右子树位置，由于不能连续的红色，所以其父结点3和叔结点2都变为黑色，祖父结点变为红色，由于祖父结点2为根结点，所以重新变成黑色。

### *红黑树删除概述*

红黑树的播入操作容易导致连续的两个红结点，破坏性质④。而删除操作容易造成子树黑高的变化（删除黑结点会导致根结点到叶结点间的黑结点数量减少），破坏性质⑤。

所以红黑树删除要比红黑树插入更复杂。

删除过程也是先执行二叉查找树的删除方法。若待删结点有两个孩子，不能直接删除，而要找到该结点的中序后继（或前驱）填补，即右子树中最小的结点，然后转换为删除该后继结点。由于后继结点至多只有一个孩子，这样就转换为待删结点是叶结点或仅有一个孩子的情况。

- 前驱结点为二叉查找树中序遍历中当前点的前一个结点，往往在其左子树的最右测。

- 后继结点为二叉查找树中序遍历中当前点的后一个结点，往往在其右子树的最左测。

对于二叉查找树删除而言有三种情况：

1. 被删除结点无子结点， 直接删除。
2. 被删除结点只有一个子结点，用子结点替换要被删除的结点。
3. 被删除结点有两个子结点，可以用前驱结点或者后继结点进行替换删除操作。

第一种情况不用多考虑，融合后两种情况并结合红黑树特性分为三种情况。

以完成所有插入后的红黑树为例。

### *填充结点红*

即删除后用来填补该位置的结点为红色结点。

此时无论当前结点是什么颜色，删除后不影响黑高，所以不影响红黑树平衡。

1. 当前结点为红：删除4，后继结点5为红结点，所以直接5替换4的位置并变红，其他不变。
2. 当前结点为黑：删除6，后继结点7为红结点，所以直接7替换6的位置并变黑，其他不变。

### *填充结点黑且为左结点*

即删除后用来填补该位置的结点为黑色结点，且该结点为其父结点的左子结点。

1. 填充结点的兄弟结点是红结点：删除2，前驱结点1为黑结点，其兄弟结点4为红结点。
2. 填充结点的兄弟结点是黑色，同时兄弟结点的右子结点是红色的，左子结点任意颜色：删除4，前驱结点3为黑结点，其兄弟结点6为黑结点，6的右子结点7为红结点。

### *填充结点黑且为右结点*

即删除后用来填补该位置的结点为黑色结点，且该结点为其父结点的右子结点。

### *红黑树与二叉查找树*

由二叉查找树树的“高度平衡”，降低到红黑树的“任一结点左右子树的高度，相差不超过两倍”的“适度平衡”，也降低了动态操作时调整的频率。对于一棵动态查找树，如果插入和删除操作比较少，查找操作比较多，采用$AVL$树比较合适，否则采用红黑树更合适。

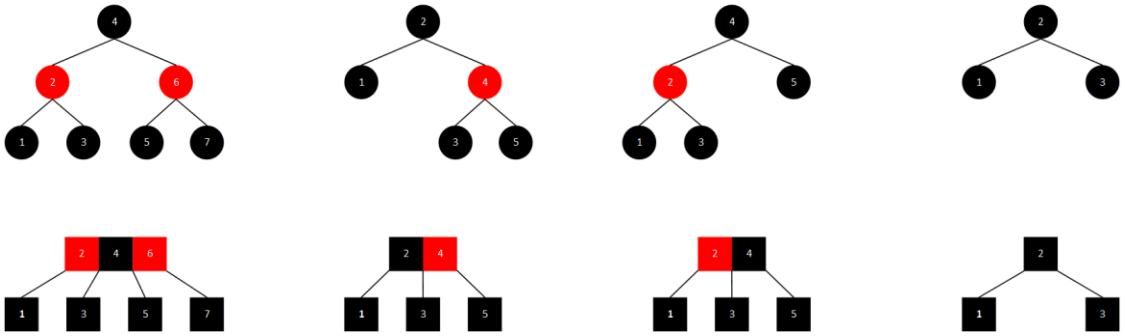
二叉查找树使用平衡因子来约束树高，而红黑树用黑高相同约束树高。

二叉查找树只要平衡因子不超过1的绝对值就不用每次调整树整体，而红黑树每次插入货删除都需要进行对树进行调整。

### *红黑树与 B 树*

红黑树在被发明之初被称为平衡二叉$B$树，所以红黑树与$B$树之间必然有联系。

- 将红黑树的所有红色结点上移到和他们的父结点同一高度上组成一个$B$树结点，就会得到一棵四阶$B$树。
- 红黑树的黑色结点个数与四阶$B$树的结点总个数相等。
- 在所有的$B$树结点中，黑色结点是父结点，红色结点是子结点。黑色结点在中间，红色结点在两边。



*red-black-b*

## 树表查找

### B 树

即多路平衡查找树，要求掌握基本特定点、操作。

### *B 树定义*

为了保证$m$叉查找树中每个结点都能被有效利用，避免大量结点浪费导致树高过大，所以规定$m$叉查找树中，除了根结点以外（根结点最小为1）：

- 任何结点至少有$\lceil\frac{m}{2}\rceil$个分叉，即至少包含$\lceil\frac{m}{2}\rceil-1$个结点。
- 至少有两棵子树。

为了保证$m$叉查找树是一棵平衡树，避免树偏重导致树高过大，所以规定$m$叉查找树中任何一个结点，其所有子树的高度都要相同。

而能保证这两点的查找树，就是一棵$B$树，即多路平衡查找树，多少叉，就是一棵多少阶的$B$树。

非叶结点定义：$\{n, P_0, K_1, P_1, \cdots, K_n, P_n\}$。其中$K_i$为结点关键字，$K_1 < K_2 < \cdots < K_n$，$P_i$为指向子树根结点的指针。$P_{i-1}$所指子树所有结点的关键字均小于$K_i$，$P_i$所指子树的关键字均大于$K_i$。

### *B 树性质*

- 树的每个结点至多包含$m$棵子树，至多包含$m-1$个关键字。
- 若根结点不是终端结点，则至少有两颗子树，有一个关键字。
- 任意结点的每棵子树都是绝对平衡的。所有结点的平衡因子均等于0的。
- 除根结点以外的所有非叶结点至少有$\lceil\frac{m}{2}\rceil$棵子树，即至少包含$\lceil\frac{m}{2}\rceil-1$个结点。
- 每个结点中的关键字是有序的。子树0<子树1<子树2<……。
- 所有叶结点都出现在同一个层次上且不带信息。（可以视为外部结点或类似于折半查 找判定树的查找失败结点，实际上这些结点不存在，指向这些结点的指针为空）。
- $B$树最底端的失败的不存在的结点就是常说的叶子结点，而最底端的存在数据的结点就是终端结点。（一般的树的叶子结点和终端结点都是指最底端的有数据的结点）
- 携带数据的是内部结点，最底部的叶子结点也称为外部结点。

$B$树相关性质计算涉及多个单元，注意一定要区分：

- 树高（默认是不包含无数据的叶子结点）。
- 子树棵数。
- 关键字。
- 结点（结点数往往小于关键字数）。

### *B 树树高与关键字*

$B$树中的大部分操作所需的磁盘存取次数与$B$树的高度成正比。高度一般与关键字相关。

计算$B$树高度大部分不包括叶子结点。若含有$n$个关键字的$m$阶$B$树。

- 最小高度：让每个结点尽可能满，有$m-1$个关键字，$m$个分叉，则一共有$(m-1)(m^0+m^1+m^2+\cdots+m^{h-1})$个结点，其中$n$小于等于这个值，从而求出$h\geq\log_m(n+1)$。
- 最大高度：
  - 让各层分叉尽可能少，即根结点只有两个分叉，其他结点只有$\lceil\frac{m}{2}\rceil$个分叉，所以第一层1个，第二层2个，第$h$层$2\left(\lceil\frac{m}{2}\rceil\right)^{h-2}$个结点，而$h+1$层的叶子结点有$2\left(\lceil\frac{m}{2}\rceil\right)^{h-1}$个，且$n$个关键字的$B$树必然有$n+1$个叶子结点，从而$n+1\geq 2\left(\lceil\frac{m}{2}\rceil\right)^{h-1}$，即$h\leq\log_{\lceil\frac{m}{2}\rceil}\frac{n+1}{2}+1$。
  - 让各层关键字尽可能少，记$k=\lceil\frac{m}{2}\rceil$。第一层最少结点数和最少关键字为1；第二层最少结点数为2，最少关键字为$2(k-1)$，第三层最少结点数为$2k$，最少关键字为$2k(k-1)$，第$h$层最少结点数为$2k^{h-2}$，

最少关键字为$2k^{h-2}(k-1)$，从而$h$层的$m$阶$B$树至少包含关键字总数$1+2(k-1)(k^0+k^1+\cdots+k^{h-2})=1+2(k^{h-1}-1)$，若关键字总数小于这个值，则高度一定小于$h$，所以$n \geq 1+2(k^{h-1}-1)$，则$h \leq \log_{\lceil \frac{m}{2} \rceil} \frac{n+1}{2}+1$。

### *B 树关键字与结点*

树高与结点：

- 对于已知树高和阶求最大最小关键字数量就是上面公式的逆运算。已知树高可以求出关键字数量。
- 根据关键字数量和结点之间的关系相除。求最大高度就除以每结点最小关键字数，求最小高度就除以每结点最大关键字数。

关键字与结点：

- 具有$n$个关键字的$m$阶$B$树，应有$n+1$个叶结点。

叶结点即查询失败的结点，对于$n$个关键字查找则可能的失败范围有$n+1$种。

- 有$n$个非叶结点的$m$阶$B$树中至少包含$(n-1)\left(\lceil \frac{m}{2} \rceil-1\right)+1$个关键字。

除根结点外的$n-1$个$m$阶$B$树中的每个非叶结点最少有$\lceil \frac{m}{2} \rceil-1$，然后再加上根结点的一个，所以最少为$(n-1)\left(\lceil \frac{m}{2} \rceil-1\right)+1$个。

### *B 树查找*

$B$树的查找包含两个基本操作：

1. 在$B$树中找结点。
2. 在结点内找关键字。

由于$B$树常存储在磁盘上，因此前一个查找操作是在磁盘上进行的，而后一个查找操作是在内存中进行的，即在找到目标结点后，先将结点信息读入内存，然后在结点内采用顺序查找法或折半查找法。

在$B$树上查找到某个结点后，先在有序表中进行查找，若找到则查找成功，否则按照对应的指针信息到所指的子树中去查找，则说明树中没有对应的关键字，查找失败。

### *B 树插入*

新元素插入一定是插入到最底层的终端结点，使用$B$树的查找来确定插入位置。插入位置一定是最底层的某个非叶结点。

若导致原结点关键字数量超过上限溢出（$m-1$个关键字），就从中间位置$\lceil\frac{m}{2}\rceil$分开，将左部分包含的关键字放在原来结点，右部分包含的关键字放在一个新结点，并插入到原结点的父结点的后一个位置上，而在原结点的父结点连接后的结点后移一个连接让位给分割出来的右半部分结点，中间的一个结点$\lceil\frac{m}{2}\rceil$插入到原结点的父结点上，并考虑在父结点的顺序对指针进行调整保证顺序。

若父结点插入时也溢出了，则同理在父结点的中间进行分割，左半部分在原来父结点；右半部分新建一个父结点，并把中间结点右边开始的所有连接移动到新父结点上；中间的结点上移到祖父结点，如果没有就新建，然后建立两个指针分别指向原父结点和新父结点。

### *B 树删除*

若被删除关键字在终端结点，且结点关键字个数不低于下限，则直接删除该关键字，并移动后面的关键字。

若被删除关键字在非终端结点，则用直接前驱或直接后继来替代被删除关键字，然后后面的元素直接前移：

- 直接前驱：当前关键字左侧指针所指子树遍历到最右下的元素。
- 直接后继：当前关键字右侧指针所指子树遍历到最左下的元素。

若被删除关键字在终端结点，但是结点关键字个数删除后低于下限：

- 右兄弟够借：若原结点右兄弟结点里的关键字在删除一个后高于下限，则可以用结点的后继以及后继的后继来顶替：
    1. 将原结点在父结点的连接的后一个关键字（后继元素）下移到原结点并放在最后面。
    2. 将原结点右兄弟结点的第一个关键字上移插入到下移的元素的空位。
    3. 原结点右兄弟结点里的关键字全部前移一位。
- 左兄弟够借：若原结点里右兄弟的关键字在删除一个后低于下限，但是左兄弟的结点足够，则可以用结点的前驱以及前驱的前驱来顶替：
    1. 将原结点在父结点的连接的前一个关键字（前驱元素）下移到原结点并放在最前面，其余元素后移。
    2. 将原结点左兄弟结点的最后一个关键字上移插入到原结点父结点的连接的前面。
    3. 原结点左兄弟结点里的关键字全部前移一位。
- 左右兄弟都不够借：若左右兄弟结点的关键字个数均等于下限值，则将关键字删除后与左或右兄弟结点以及父结点中的关键字进行合并：
    1. 将原结点的父结点连接后的关键字插入到原结点关键字最后面。
    2. 将原结点的左或右兄弟结点的关键字合并到原结点（前插或后插），并将连接也转移到原结点上。

3. 若父结点的关键字个数又不满于下限，则父结点同样要于与它的兄弟父结点进行合并，并不断重复这个过程。
4. 若父结点为空则删除父结点。（兄弟合并，父亲下沉）

## B+树

$B+$树考的并不是很深。用于数据库。

与分块查找的思想类似，是对$B$树的一种变型，多用于索引结构。

### *B+树定义*

一个$m$阶的$B+$树需要满足以下条件：

1. 每个分支结点最多有$m$棵子树或孩子结点。
2. 为了保持绝对平衡，非叶根结点至少有两棵子树，其他每个分支结点至少有$\lceil\frac{m}{2}\rceil$棵子树。（不同于$B$树，$B+$树又重新将最下面的保存的数据定义为叶子结点）
3. 结点的子树个数与关键字个数相等。（$B$树结点子为树个数与关键字个数加1）
4. 所有叶结点包含所有关键字以及指向记录的指针，叶结点中将关键字按大小排序，并且相邻叶子结点按大小顺序相互连接起来。所以$B+$树支持顺序查找。
5. 所有分支结点中仅包含其各子结点中关键字的最大值以及指向其子结点的指针（即分支结点只是索引）。

### *B+树查找*

无论查找成功与否，$B+$树的查找一定会走到最下面一层结点，因为对应的信息指针都在最下面的结点。而$B$树查找可以停留在任何一层。

$B+$树可以遍历查找，即从根结点出发，对比每个结点的关键字值，若目标值小于当前关键字值且大于前一个关键字值，则从当前关键字的指针向下查找。

$B+$树可以顺序查找，在叶子结点的块之间定义指向后面叶子结点块的指针，从而能顺序查找。

### *B+树与 B 树区别*

对于$m$阶$B+$树与$B$树：

| | B+树 | B 树 |
|---|---|---|
| 结点的 n 个关键字对应的子 | $n$ | $n+1$ |

| | B+树 | B 树 |
|---|---|---|
| 树个数 | | |
| 根结点的关键字数 | $[1,m]$ | $[1,m-1]$ |
| 其他结点的关键字数 | $\left[\lceil\frac{m}{2}\rceil,m\right]$ | $\left[\lceil\frac{m}{2}\rceil-1,m-1\right]$ |
| 关键字分布 | 叶子结点包含所有关键字，非叶结点包含部分重复关键字 | 所有结点的关键字不重复 |
| 结点作用 | 叶子结点包含信息，非叶子结点是索引作用 | 所有结点都包含信息 |
| 结点存储内容 | 叶子结点包含关键字与对应记录的存储地址，非叶子结点包含对应子树的最大关键字和指向该子树的指针 | 所有结点都包含关键字与对应记录存储地址 |
| 查找方式 | 随机查找、顺序查找 | 随机查找 |
| 查找位置 | 需要查找到叶子结点的最底层才能判断是否查找成功或失败 | 查找到数的任何地方都能判断 |
| 查找速度 | 非叶子结点不包含关键字对应记录的存储地址，可以使一个磁盘块含有多格关键字，从而让树的阶数更大，树更矮，读磁盘次数更是，查找更快 | 所有结点都包含存储地址，保存的关键字数量更少，树高更高，所以读写磁盘次数更多，查找更慢 |

## 散列表查找

线性表和树表中，数据位置与数据关键字无关，而散列表数据位置与关键字有关。

### 散列表定义

散列表又称哈希表，是一种数据结构，数据元素的关键字与其存储地址直接相关。一个散列结构是一块地址连续的存储空间。

### 散列函数

关键字与地址通过散列函数（哈希函数）来实现映射。即记录位置=散列函数(记录关键字)。

- 直接定址法：可表示为散列函数(关键字)=$a$ ×关键字+$b$，其中$a$、$b$均为常数。这种方法计算特别简单，并且不会发生冲突，但当关键字分布不连续时，会出现很多空闲单元，会将造成大量存贮单元的浪费。

- 数字分析法：分析关键字的各个位的构成，截取其中若干位作为散列函数值，尽可能使关键字具有大的敏感度，即最能进行区分的关键字位，这些位数都是连续的。
- 平方取中法：先求关键字的平方值，然后在平方值中取中间几位为散列函数的值。因为一个数平方后的中间几位和原数的每一位都相关，因此，使用随机分布的关键字得到的记录的存储位置也是随机的。适用于关键字的每位取值都不够均匀或均小于散列地址所需的位数。
- 折叠法：将关键字分割成位数相同的几部分(最后一部分的位数可以不同)，然后取这几部分的叠加和(舍去进位)作为散列函数的值。例如，假设关键字为某人身份证号码$430\,1046\,8101\,5355$，则可以用4位为一组进行叠加，即有$5355 + 8101 + 1046 + 430 = 14932$，舍去高位，则有$H(430104681015355) = 4932$。
- 随机数法：对于存储位置给定随机数安排，查找起来会很麻烦。
- 除留余数法：散列函数(关键字)=关键字%$p$，$p$一般是不大于表长的最大质数。这种方法使用较多，关键是选取较理想的$p$值，使得每一个关键字通过该函数转换后映射到散列空间上任一地址的概率都相等，从而尽可能减少发生冲突的可能性。一般情形下，取$p$为一个素数较理想，如果是合数则因为可以被多个数整除从而多个关键字余数相同造成冲突。

### 映射冲突

一般情况下，设计出的散列函数很难是单射的，即不同的关键字对应到同一个存储位置，这样就造成了冲突（碰撞）。此时，发生冲突的关键字互为同义词。

#### *开放定址法*

可存放新表项的空闲地址既向同义词开放也向非同义词开放。从发生冲突的那个单元开始，按照一定的次序，从哈希表中找出一个空闲的存储单元，把发生冲突的待插入关键字存储到该单元中，从而解决冲突。既指如果当前冲突，则将元素移动到其他空闲的地方。

$H_i = (H(key) + d_i) \bmod m$。

- $i$表示发生第$i$次冲突，$i = 1,2,\cdots,m-1)$。
- $m$为散列表长度，类似于循环队列，超出表长以后就循环到最左边。
- $d_i$为增量序列，是指发生第$i$次冲突的时候，$H(key)$偏移了多少位。

增量序列选择：

- 线性探测法：$d_i = 1,2,3,\cdots,m-1$。线性探测法充分利用了哈希表的空间，但在解决一个冲突时，可能造成新的冲突。

- 二次（平方）探测法：$di = 1, -1, 2^2, -2^2 \cdots, \left(\frac{m}{2}\right)^2, -\left(\frac{m}{2}\right)^2$。对比线性探测法更不容易产生聚集问题。注意：散列表长度$m$必须是一个可以表示为$4j + 3$的素数才能探测到所有位置。
- 伪随机探测法：定义$d_i$是一个伪随机数。
- 再散列法：$d_i = Hash_2(key)$，又称双散列法。需要使用两个散列函数,当通过第一个散列函数$H(key)$得到的地址发生冲突时，则利用第二个散列函数$Hash_2(key)$计算该关键字的地址增量。它的具体散列函数形式：$H_i = \big(H(key) + i \times Hash_2(key)\big)\%m$。初始探测位置$H = H(key)\%m$。$i$是冲突的次数，初始为$0$。在再散列法中，最多经过$m - 1$次探测就会遍历表中所有位置，回到$H_0$位置。

**聚集**：同义和非同义关键字都堆积到一起。原因是选取不当的处理冲突的方法。对查找长度有直接的影响。

注意：在开放定址的情形下，不能随便物理删除表中的已有元素，因为若删除元素，则会截断其他具有相同散列地址的元素的查找地址。因此，要删除一个元素时，可给它做一个删除标记，进行逻辑删除。但这样做的副作用是:执行多次删除后，表面上看起来散列表很满，实际上有许多位置未利用，因此需要定期维护散列表，要把删除标记的元素物理删除。

### *链地址法*

又称为拉链法或链接法，是把相互发生冲突的同义词用一个单链表链接起来，若干组同义词可以组成若干个单链表。思想类似于邻接表的基本思想，且这种方法适合于冲突比较严重的情况。

指针需要额外的空间，故当结点规模较小时，开放定址法较为节省空间，而若将节省的指针空间用来扩大散列表的规模，可使装填因子变小，这又减少了开放定址法中的冲突，从而提高平均查找速度。

每次冲突都要重新哈希，计算时间增加。

### *公共溢出区法*

为所有冲突的关键字记录建立一个公共的溢出区来存放。在查找时，对给定关键字通过散列函数计算出散列地址后，先与基本表的相应位置进行比对，如果相等，则查找成功；如果不相等，则到溢出表进行顺序查找。如果相对于基本表而言，在有冲突的数据很少的情况下，公共溢出区的结构对查找性能来说还是非常高的。

## 散列查找

先通过散列函数计算目标元素存储地址，然后根据解决冲突的方法进行下一步的查询。

如果使用拉链法通过散列函数计算得到存储地址为空，则可以直接代表查找失败，这时候一般定义查找长度这里不算。

而如果使用开放地址法计算得到空位置的时候，代表查找失败，但是这时候需要定义查找长度要算这个地址。

若散列函数设计得足够好，散列查找时间复杂度可以达到$O(1)$，即不存在冲突。

散列表的查找效率取决于三个因素：散列函数、处理冲突的方法和装填因子。

**装填因子**=表中记录数÷散列表长度。装填因子代表一个散列表中的满余情况，越大则查找效率越低。

若只给出了装填因子$\alpha$，则此时平均查找长度为：$ASL = \frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right)$。

<!doctype html>

9-sort

排序

基本概念

排序：将一个数据元素的任意序列重新排列成一个按关键字有序的序列。

内部排序：待排序的记录存放在计算机的内存中所进行的排序操作称为内部排序。

外部排序：待排序的记录数量很大，以致内存一次不能容纳全部记录，在排序过程中需要访问外存的排序过程称为外部排序。

稳定的排序：比如一个序列是“1,4,3,3,*2”，按从小到大排序后变成“1,2,3,3*,4”，就叫做稳定排序，即 3 和 3*相对顺序不变。如果相同关键字的顺序发生了改变，则是不稳定的排序。稳定性的需要看具体的应用场景。

内部排序的算法性能取决于算法的时间复杂度和空间复杂度，而时间复杂度一般是由比较和移动次数决定的。外部排序除此之外还要考虑磁盘读写速度和次数。

大部分排序算法都仅适用于顺序存储的线性表。

大部分排序需要经过比较和移动两个过程。（基数排序不需要比较）排序时至少需要比较_2(n!)次。每次比较两个关键字后，仅出现两种可能的转移。假设整个排序过程至少需要做 t 次比较。则显然会有 $2^t$ 种情况。由于 n 个记录共有 n!种不同的排列，因而必须有 n!种不同的比较路径，于是有 $2^t$n!，所以得到不等式。

插入排序

插入排序的排序序列分为非排序和已排序序列。插入排序就是将选定的目标值插入到对应的位置。

直接插入排序

直接插入排序过程

每次将一个待排序的记录按其关键字大小插入到前面已排好序的子序列中，直到全部记录插入完成为止。

直接插入排序性能

空间复杂度为 O(1)。

时间复杂度主要来自对比关键字，移动元素，若有 n 个元素，则需要 n-1 趟处理。

最好情况是原本的序列就是有序的，需要 n-1 趟处理，每次只需要对比一次关键字，不用移动元素，时间复杂度为 O(n)。

最坏情况是原本的序列是逆序的，需要 n-1 趟处理，第 i 趟处理需要对比关键字 i+1 次，移动元素 i+2 次，时间复杂度是 O(n^2)。

所以平均时间复杂度是 O(n^2)。

直接插入排序算法是稳定的。

如果使用链表实现直接插入排序，移动元素的次数变少了，但是关键字对比次数仍然时 $O(n^{2)}$，从而整体时间复杂度依然是 O(n2)。

直接插入排序特性

在待排序的元素序列基本有序的前提下，直接插入排序效率最高的。

直接插入排序进行 n 躺后能保证前 n+1 个元素是有序的，但是不能保证其都在最终的位置上（如最小的元素在最后一位，这是插入排序的共性）。

二分插入排序

二分插入排序过程

也称为折半插入排序，是对直接插入排序的优化，在寻找插入位置时使用二分查找的方式。

当 data[mid]==data[i]时，为了保证算法的稳定性，会继续在 mid 所指位置右边寻找插入位置。

当 low>high 时停止折半查找，并将[low,i-1]内的元素全部右移，并把元素值赋值到 low 所指的位置。

折半插入排序是找到位置了后一起移动元素，而直接插入排序是边查找边排序。

二分插入排序性能

空间复杂度为 O(1)。

二分插入排序是稳定的。

比起直接插入排序，比较关键字的次数减少为 O(n_2n)，移动元素的次数没变，所以总体时间复杂度为 O(n^2)。

二分插入排序特性

对于直接插入的优化仅在于二分查找的比较次数。

二分插入排序进行 n 躺后能保证前 n+1 个元素是有序的，但是不能保证其都在最终的位置上（如最小的元素在最后一位，这是插入排序的共性）。

希尔排序

又称缩小增量排序。

希尔排序过程

希尔排序也是对直接插入排序的优化。直接插入排序对于基本有序的序列排序效果较好，所以就希望序列能尽可能基本有序。从而希尔排序的思想就是先追求表中元素部分有序，然后逐渐逼近全局有序。

先将整个待排序元素序列分割成若干个子序列（由相隔某个“增量”的元素组成的），分别进行直接插入排序，然后缩小增量重复上述过程，直到增量为 1。每次对比只对比两个以上的个元素进行插入交换。

增量序列的选择建议是第一趟选择元素个数的一半，后面不断缩小到原来的一半。

希尔排序性能

空间复杂度为 O(1)。

而时间复杂度和增量序列的选择有关，目前无法使用属性手段证明确切的时间复杂度。最坏时间复杂度为 $O(n^{2)}$，在某个范围内可以达到 $O(n\{1.3\})$。

希尔排序是不稳定的。（因为相同的元素可能分到不同的子序列中进行重排打乱原有顺序）

希尔排序只适用于顺序表而不适合用于链表，无法快速进行增量的访问。

希尔排序特性

希尔排序在最后一趟前都不能保证元素在最后的位置上。

希尔排序在最后一趟前都不能保证元素是有序的。

交换排序

交换排序即根据序列中两个元素关键的比较结构然后交换这两个记录在序列中的位置。

冒泡排序

重点就是相邻两两比较。

冒泡排序过程

从后往前或从前往后两两比较相邻元素的值，若逆序则交换这两个值，如果相等也不交换，直到序列比较完。这个过程是一趟冒泡排序，第 i 趟后第 i 个元素会已经排序完成。每一趟都会让关键字最小或最大的一个元素到未排序队列的第一个或最后一个。一共需要 n-1 趟排序。

冒泡排序中所产生的有序子序列一定是全局有序的（不同于直接插入排序），也就是说，有序子序列中的所有元素的关键字一定小于或大于无序子序列中所有元素的关键字，这样每趟排序都会将一个元素放置到其最终的位置上。

冒泡排序性能

空间复杂度为 O(1)。

最好情况下即本身序列有序，则比较次数是 n-1，交换次数是 0，从而时间复杂度是 O(n)。

最坏情况是逆序情况，比较次数和交换次数都是，所以时间复杂度是 O(n^2)。

从而平均时间复杂度是 O(n^2)。

冒泡排序是稳定的。

冒泡排序可以用于链表。

冒泡优化

对于每行冒泡进行优化：如果发现排序前几轮就已经实现了排序成功，那么后面的排序岂不是都浪费了时间进行比较？可以在第一轮循环中设置一个布尔值为 false，如果在这一轮发生排序交换就设置为 true，如果一轮结束后发现这个值还是 false，说明这一轮没有进行交换，表示已经排序成功，就直接所有退出循环。

对于每列冒泡进行优化：默认每一轮冒泡是从[length-i]结束，如一共 5 个元素排序，需要 4 轮排序，第二轮冒泡排序应该从 0 开始，到 3 结束，因为最后一个元素 4 已经在第一轮排序成功。但是如果在第二轮发现 2，3 已经排序成功了不需要交换，那么默认排序方法第三轮还是要从 0 到 2 进行排序，还要比较一次 1 和 2 位置的数据，这就造成了浪费，那么如何解决？记录每一轮发生比较的元素的最大索引值，下一轮比较到这个索引值直接结束，不需要继续比较后面的元素。如果最大索引值为 0 则直接退出。这就进一步优化了上面一种策略。

冒泡排序特性

冒泡排序产生的序列全局有序，n 趟排序后第 n 个元素到达最终的位置上，前 n 个或后 n 个位置的元素确定。

快速排序

排序过程类似于构建二叉排序树。基于分治法。

快速排序过程

取待排序序列中的某个元素 pivot 作为基准（一般取第一个元素），通过一趟排序，将待排元素分为左右两个子序列，左子序列元素的关键字均小于或等于基准元素的关键字，右子序列的关键字则大于基准元素的关键字，称进行了一趟快速排序（一次划分），这个 pivot 已经成功排序。然后分别对两个子序列继续进行快速排序，直至整个序列有序。

单边循环快排

即 lomuto 洛穆托分区方案。

选择最右边的元素值做标杆，把标杆值放入 pivot 变量中。

初始时，令 low 和 high 都指向最左边的元素。其中 low 用于被动向右移动，维护小于标杆值的元素的边界，即每次交换的目标索引，一旦交换 low 就向右移动一个；high 用于主动向右移动，寻找比标杆值小的元素，一旦找到就与 low 指向元素进行交换。

然后 high 开始移动，判断 high 指向的元素值是否小于 pivot 值，如果不小于就继续向右移动。

当遇到比标杆小的值，high 指向的值就和 low 指向的值进行交换，如果 high 和 low 指向的值为同一个则不进行交换，然后 low 右移一个，high 继续右移查找。

high 继续移动，最后 high>=pivot 时将基准点元素值与 low 指向值进行交换，该轮排序结束。此时 low 指向的位置就是 pivot 值所应该在的位置。

返回基准点元素所在索引，从而确定排序上下边界，递归继续执行排序。

双边循环快排

分为普通分区方案和 hoare 霍尔分区方案。逻辑基本上一样，只是边界选择方式不同。

先选择个值做标杆，一般为最左边值，把标杆值放入 pivot 变量中。

初始时，令 high 指向序列最右边的值，low 指向序列最左边的值。

然后从 high 开始不断左移，当遇到比标杆大或等于的值时 high–。

如果发现比标杆小的值，即 high<pivot，需要交换，然后 high 不动，low++开始移动去找要交换的大于标杆的值。

若 low 所指向的值比标杆小或等于，则 low++进行寻找。（为什么要加上一个等于标杆值可以继续向右移动的条件？因为标杆值默认是第一个值，即初始化 pivot 跟 low 指向同一个值，如果只能小于标杆值才能继续移动不交换，则在第一个元素时，由于标杆值被初始化赋值为第一个值，则标杆 pivot 值等于 low 值，从而导致 low 值跟 high 值进行交换，导致 pivot 标杆值被交换走了，标杆值变为了最开始最右边的 high 值，导致了排序有问题）

若 low 所指向的值比标杆大，则 low 值与 high 值进行交换。

交换后 low 不动，high–开始移动。回到步骤三开始执行。

当 low=high 时表示 low 和 high 之前的元素都比基准小，low 和 high 之后的元素都比基准大，完成了一次划分。然后把基准元素放入 low==high 指向的位置。

不断交替使用 low 和 high 指针进行对比。对左右子序列进行同样的递归操作即可，从步骤三开始。若左右两个子序列的元素数量小等于一，则无需再划分。

即对序列进行比较，有头尾两个指针，尾指针开始比较向前移动，若指向值比对比值小则要交换，交替让头指针开始移动，否则不改变指针则尾指针继续向前；同理头指针向后移动，若指向值比对比值大则交换，交替让尾指针移动，否则不改变指针则头指针继续向后。最后头尾指针指向一个位置，将对比值插入到当前值，此时一趟完成。

洛穆托分区与霍尔分区比较。

快速排序性能

由于快速排序使用了递归，所以需要递归工作栈，空间复杂度与递归层数相关，所以为 O(递归层数)。

每一层划分只需要处理剩余的待排序元素，时间复杂度不超过 O(n)，所以时间复杂度为 O(n 归层数)。

而快速排序会将所有元素组织成为二叉树，二叉树的层数就是递归调用的层数。所以对于 n 个结点的二叉树，最小高度为_2n+1，最大高度为 n。

从而最好时间复杂度为 O(n_2n)，最坏时间复杂度为 O(n^2)，平均时间复杂度为 O(n_2n)；最好空间复杂度为 O(_2n)，最坏空间复杂度为 O(n)，平均空间复杂度为 O(_2n)。

所以如果初始序列是有序的或逆序的，则快速排序性能最差（速度最慢）。若每一次选中的基准能均匀划分，尽量让数轴元素平分，则效率最高（速度最快）。性能与分区处理顺序无关。

所以对于快速排序性能优化是选择尽可能能中分的基准元素，入选头中尾三个位置的元素，选择中间值作为基准元素，或随机选择一个元素作为基准元素。

最好使用顺序存储，这样找到数轴元素与遍历时比较简单。

快速排序算法是不稳定的。

快速排序特性

快速排序不产生有序子序列。

枢轴元素到达的位置是不确定的，但是每次都会到其最终的位置上。第 n 趟有 n 个元素到最终位置上。

求快速排序趟数就是找到符合这种性质的元素个数。

快速排序在内部排序中的表现最好。

对于基本有序或倒序的序列，快速排序速度最慢。

对于每次的数轴元素能尽量将表分为长度相同的子表，快速排序速度最快。

选择排序

分为已排序和未排序序列。选择排序就是每一趟在待排序元素中选取关键字最小或最大的元素加入有序子序列。

选择排序也需要交换，但是与交换排序的不断交换不同的是选择排序时选择出一个最后进行交换，一趟只交换一次。

选择排序也需要插入，且也分为已排序和未排序序列，但是插入排序不需要选择，且元素移动方式是插入而不是交换。

选择排序算法的比较次数始终为 n(n-1)/2，与序列状态无关。

简单选择排序

简单选择排序过程

即每一趟在待排序元素中选取关键字最小的元素加入有序序列。交换发生在选出最值后，在每趟的尾部。经过 n-1 趟就可以完成。

简单选择排序性能

空间复杂度为 O(1)。

时间复杂度为 O(n^2)。

简单选择排序是不稳定的。因为选择后会进行交换，影响顺序。

简单选择排序也可以适用于链表。

直接插入排序与简单选择排序

插入排序和选择排序都是分为未排序和已排序两个部分，那么其中有什么区别？

如 18、23、19、9、23*、15 进行排序。

18 23 19 9 23* 15

插入排序：

选择排序：

堆排序

堆的定义

若 n 个关键字序列 L 满足下面某一条性质，则就是堆：

若满足 L(i)L(2i)且 L(i)L(2i+1),(1i)则是大根堆或大顶堆。

若满足 L(i)L(2i)且 L(i)L(2i+1),(1i)则是小根堆或小顶堆。

所以堆就是用顺序存储的完全二叉树。

堆的叶子结点范围是_2n+1n。

堆的建立

其实堆就是层序存储的完全二叉树。其中：

i 的结点都是非终端结点。

i 的左孩子是 2i。

i 的右孩子是 2i+1。

i 的父结点是。

所以建立根堆过程是：

按照关键字序列依次添加关键字到二叉树，按照层次遍历顺序添加。

初始化成功后再从下往上、从左至右按逆层次遍历顺序不断调整位置。

如果是大根堆则大元素往上，且当前结点与更大的孩子结点互换；如果是小根堆则小元素往上，且当前结点与更小的孩子结点互换。

递归往上时父子结点不断交换位置。

如果元素互换破坏了调整好的下一级的堆，则使用同样的方法对下一层递归调整。

如用堆排序对(15,9,7,8,20,-1,7,4)建立小根堆堆。首先将这组数据按层序初始化为无序堆，然后从最后向前开始调整：

<!– 1. 从 t<的结点开始往前遍历。

检查当前结点 i 与左孩子和右孩子是否满足根堆条件，若不满足则交换。

若是建立大根堆，检查是否满足根大于等于左、右结点，若不满足，则当前结点与更大的一个孩子互换。

若是建立小根堆，检查是否满足根小于等于左、右结点，若不满足，则当前结点与更小的一个孩子互换。

若元素互换破坏了下一级的堆，则采用同样的方法继续向下调整。

若是建立大根堆，则小的元素不断下坠。

若是建立小根堆，则大的元素不断下坠。 –>

堆排序过程

由于选择排序是在每一趟都选择最大或最小的值进行排序，所以堆排序中就通过堆这个存储结构来完成对最值的选取——直接选择堆顶元素。

堆排序即每次将堆顶元素与堆底元素（堆最底层最右元素）进行交换，表示这个部分已经排序完成了不需要进行调整，第 i 趟表示倒数 i 个元素已经有序，所以无序的元素就是堆前面的元素。

每一趟将堆顶元素加入子序列（堆顶元素与待排序序列中的最后一个元素交换）。此时后面的这个元素就排序好了。最右下的元素作为堆顶元素。

此时待排序序列已经不是堆了（堆顶不能保证是最小或最大的元素），需要将其再次调整为堆（小元素或大元素不断下坠）。

重复步骤一二。

直到 n-1 趟处理后得到有序序列。基于大根堆的堆排序会得到递增序列，而基于小根堆的堆排序会得到递减序列。

调整堆从右边即序列末尾开始。

堆排序性能

堆排序的存储就是它本身，不需要额外的存储空间，要么只需要一个用于交换或临时存放元素的辅助空间。所以空间复杂度为 O(1)。

若树高为 h，某结点在第 i 层，则将这个结点向下调整最多只需要下坠 h-i 层，关键字对比次数不超过 2(h-i)次。

第 i 层最多 2^{i-1}个结点，而只有第 1(h-1)层的结点才可能需要下坠调整。所以调整时关键字对比次数不超过{i=h-1}12{i-1}2(h-i)=∑{j=1}{h-1}2^{h-j}j2n_{j=1}^{h-1}4n。

所以建堆过程中，关键字对比次数不超过 4n，建堆的时间复杂度为 O(n)。

堆排序中处理时根结点最多下坠 h-1 层，而每下坠一层，最多对比关键字两次，所以每一趟排序的时间复杂度不超过 O(h)=O(_2n)，一共 n-1 趟，所以时间复杂度为 O(n_2n)。所以总的时间复杂度也是 O(n_2n)。

堆排序是不稳定的。

堆排序适合关键字较多的情况。创如，在 1 亿个数中选出前 100 个最大值，首先使用一个大小为 100 的数组，读入前 100 个数，建立小顶堆，而后依次读入余下的教，若小于堆顶则舍弃，否则用该数取代堆顶并重新调整堆，待数据读取完毕，堆中 100 个数即为所求。

堆的插入

新元素放到表尾（即最右下角元素），并与其的父结点进行对比，若新元素比父元素更大（大根堆）或更小（小根堆），则二者互换，并保持上升，直到无法上升为止。时间复杂度为树高 O(_2n)。

堆的删除

被删除的元素用堆底元素（即最右下角元素）代替，然后让这个元素不断下坠，直到无法下坠为止。时间复杂度为树高 O(_2n)。

堆排序特性

适合大量数据进行排序。

在含有 n 个关键字的小根堆中，关键字最大的记录存储范围为+1n。这是小根堆，关键字最大的记录一定存储在这个堆所对应的完全二叉树的叶子结点中；又因为二叉树中的最后一个非叶子结点存储在中，所以得到范围。

归并排序

归并是指把两个（二路归并）或多个（多路归并）已经有序的序列合并为一个。

该算法是采用分治法的一个非常典型的应用。将已有序的子序列合并，得到完全有序的序列；即先使每个子序列有序，再使子序列段间有序。

在较大数据进行排序时为了加快速度使用归并排序，用空间换时间。

二路归并排序

二路归并排序比较常用，且基本上用于内部排序，多路排序多用于外部排序。

二路归并排序过程

把长度为 n 的输入序列分成两个长度为的子序列。

对这两个子序列分别采用归并排序。

将两个排序好的子序列合并成一个最终的排序序列。

归并排序趟数为_2n。

二路归并排序性能

二路归并排序是一棵倒立的二叉树。

空间复杂度主要来自辅助数组，所以为 O(n)，而递归调用的调用栈的空间复杂度为 O(_2n)，总的空间复杂度就是为 O(n)，无论平均还是最坏，所以这个算法在内部排序算法中空间消耗最大。

n 个元素二路归并排序，归并一共要_2n 趟，每次归并时间复杂度为 O(n)，则算法时间复杂度为 O(n_2n)

归并排序是稳定的。

分配排序

分配排序过程无须比较关键字，而是通过用额外的空间来“分配”和“收集”来实现排序，它们的时间复杂度可达到线性阶 O(n)。简言之就是：用空间换时间，所以性能与基于比较的排序才有数量级的提高。

基数排序

基数排序不是基于比较的排序算法，其核心在于将输入的数据值转化为键存储在额外开辟的数组空间中。作为一种线性时间复杂度的排序，计数排序要求输入的数据必须是有确定范围的整数。

只能对整数进行排序。

元素的移动次数与关键字的初始排列次序无关。

基数的定义

假设长度为 n 的线性表中每个结点 a_j 的关键字由 d 元组(k_j^{d-1},k_j{d-2},,k_j^{1,k_j0)组成，其中 0k_j^ir-1,(0id-1)，其中 r 就是基数。

基数排序过程

有最高位优先 MSD 和最低位优先 LSD 两种方法。

若是要得到递减序列：

初始化：设置 r 个空辅助队列 Q_{r-1},Q_{r-2},,Q_0。

按照每个关键字位权重递增的次序（个、十、百），对 d 个关键字位分别做分配和收集。

分配就是顺序扫描各个元素，若当前处理的关键字位为 x，就将元素插入 Q_x 队尾。

收集就是把 Q_{r-1},Q_{r-2},,Q_0 各个队列的结点依次出队并链接在一起。

基数排序性能

基数排序基本上使用链式存储而不是一般的顺序存储。

需要 r 个辅助队列，所以空间复杂度为 O(r)。

一趟分配 O(n)，一趟收集 O(r)，一共有 d 趟分配收集，所以总的时间复杂度为 O(d(n+r))。与序列初始状态无关。

基数排序是稳定的。

基数排序的应用

对于一般的整数排序是可以按位排序的，也可以处理一些实际问题，如根据人的年龄排序，需要从年月日三个维度分别设置年份的队列、月份的队列（1 到 12）、日的队列（1 到 31）。

所以基数排序擅长解决的问题：

数据元素的关键字可以方便地拆分为 d 组，且 d 较小。

每组关键字的取值范围不大，即 r 较小。

数据元素个数 n 较大。

计数排序

作为一种线性时间复杂度的排序，计数排序要求输入的数据必须是有确定范围的整数。

计数排序过程

找出待排序的数组中最大和最小的元素。

统计数组中每个值为 i 的元素出现的次数，存入数组 C 的第 i 项。

对所有的计数累加（从 C 中的第一个元素开始，每一项和前一项相加）。

反向填充目标数组：将每个元素 i 放在新数组的第 C(i)项，每放一个元素就将 C(i)减去 1。

当输入的元素是 n 个属于[0,k]的整数时，时间复杂度是 O(n+k)，空间复杂度也是 O(n+k)，其排序速度快于任何比较排序算法。

当 k 不是很大并且序列比较集中时，计数排序是一个很有效的排序算法。

计数排序是稳定的。

桶排序

桶排序是计数排序的升级版。它利用了函数的映射关系，高效与否的关键就在于这个映射函数的确定。桶排序的工作的原理：假设输入数据服从均匀分布，将数据分到有限数量的桶里，每个桶再分别排序（有可能再使用别的排序算法或是以递归方式继续使用桶排序进行排）。

桶排序过程

设置一个定量的数组当作空桶。

遍历输入数据，并且把数据一个一个放到对应的桶里去。

对每个不是空的桶进行排序。

从不是空的桶里把排好序的数据拼接起来。

桶排序性能

桶排序最好情况下使用线性时间 O(n)，桶排序的时间复杂度，取决与对各个桶之间数据进行排序的时间复杂度，因为其它部分的时间复杂度都为 O(n)。桶排序的平均时间复杂度为线性的 O(n+C)，其中 C=n(n-m)，其中 m 代表桶划分的数量。

很显然，桶划分的越小，各个桶之间的数据越少，排序所用的时间也会越少。但相应的空间消耗就会增大。

桶排序是稳定的。

内部排序

指在排序期间元素全部存放在内存中的排序。除了分配排序，其他的内部排序往往要经过比较和移动。

算法种类

最好时间复杂度

平均时间复杂度

最好时间复杂度

空间复杂度

是否稳定

趟数

直接插入排序

O(n)

O(n^2)

O(n^2)

O(1)

是

n-1

希尔排序

?

?

?

O(1)

否

s

简单选择排序

O(n^2)

O(n^2)

O(n^2)

O(1)

否

n-1

快速排序

O(n_2n)

O(n_2n)

O(n^2)

O(_2n)

否

初始序列

冒泡排序

O(n)

O(n^2)

O(n^2)

O(1)

是

初始序列

堆排序

O(n_2n)

O(n_2n)

O(n_2n)

O(1)

否

初始序列

二路归并排序

O(n_2n)

O(n_2n)

O(n_2n)

O(n)

是

_2n

基数排序

O(d(n+r))

O(d(n+r))

O(d(n+r))

O(r)

是

r

每趟排序结束都至少能够确定一个元素最终位置的方法：选择、交换。（插入和归并则不行）

外部排序

指在排序期间元素无法全部同时存放在内存中，必须在排序过程中根据要求不断地在内、外存之间移动的排序。

外部排序的原理

外部排序过程

磁盘的读写是以块为单位，数据读入内存后才能被修改，修改完成后还需要写回磁盘。

外部排序就是针对数据元素太多，无法一次性全部读入内存进行排序而进行处理的在外部磁盘进行的排序处理方式。

使用归并排序的方式，最少只用在内存分配三块大小的缓冲区（两个输入缓冲一个输出缓冲）即可堆任意一个大文件进行排序。然后对缓冲区里的数据进行内部排序。

外部排序过程：

生成初始归并段（大小为输入缓冲区的总大小），需要读写并进行内部排序。

重复读写，进行内部归并排序。填满输出缓冲就可以输出。输入缓冲空就可以输入新数据。

外部排序时间开销=读写外存时间（最大的时间开销）+内部排序所需时间+内部归并所需时间。

外部排序的优化方法

优化方法就是使用更多路的多路归并，减少归并趟数。

k 路平衡归并：最多只能有 k 个段归并为一个，需要一个输出缓冲区和 k 个输入缓冲区；每一趟归并中，若有 m 个归并段参与归并，则经过这一趟处理得到个新的归并段。

对 r 个初始归并段，使用 k 路归并，则归并树可以使用 k 叉树表示，若树高为 h，则归并趟数最小为 h=_kr+1。

但是多路归并会带来负面影响：

k 路归并时，需要开辟 k 个输入缓冲区，内存开销增加。

每挑选一个关键字需要对比关键字(k-1)次，内部归并时间增加。

同时，若能增加初始归并段的长度 k，也可以减少初始归并段数量 r 从而进行优化。

败者树

用于通过过去归并的经历减少归并次数。败者树可以看作一棵多了一个单个的根的完全二叉树。k 个叶结点分别是当前参加比较的元素，非叶子结点用来记忆左右子树中的失败者，而让胜者往上继续比较，一直到根结点。

如要用败者树排序 27,12,1,17,2,9,11,4，格式：元素值(归并段索引值)：

传统方法从 k 个归并段选出一个最大或最小元素需要对比关键字 k-1 次，而使用 k 路归并的败者树只需要对比关键字_2k 次（败者树层数，不包括成功结点）。

构建败者树还是需要 n-1 次对比。

置换选择排序

如果内存工作区只能容纳 l 个记录，则初始归并段也只能包含 l 条记录，若文件共有 n 条记录，则初始归并段的数量为 r=n/l。

用于构建更长的初始归并段，从而减少归并次数。

假设初始始待排文件为 FI，初始归并段输出文件为 FO，内存工作区为 WA，FO 和 WA 的初始状态为空，WA 可容纳 w 个记录。置换选择算法的步骤如下：

从 FI 输入 w 记录到工作区 WA。

从 WA 中选出其中关键字取最小值的记录，记为 MINIMAX 记录。

将 MINIMAX 记录输出到 FO 中去。

若 FI 不空，则从 FI 输入下一个记录到 WA 中。

从 WA 中所有关键字比 MINIMAX 记录的关键字大的记录中选出最小关键字记录，作为新的 MINIMAX 记录。

重复步骤三到五，如果新输入到 FI 的关键字小于 MINIMAX 的值，则驻留在 WA 中，直至在 WA 中填满选不出新的 MINIMAX 记录为止，由此得到一个初始归并段，输出一个归并段的结束标志到 FO 中去。准备输出新的归并段。

重复步骤二到六，直至 WA 为空。由此得到全部初始归并段。

此时输出的初始归并段可以超过 WA，且初始归并段长度是不一定相等的。

如 FI：4,6,9,7,13,11,14,22,30,2,3,19,20,17,1,23,5,36,12,18,21,39，WA 长度为 3，FO 为 4,6,7,9,11,13,14,16,22,30、2,3,10,17,19,20,23,36、1,5,12,18,21,39。

^表示中断当前归并段，下一个 FI 开启新的归并段；√表示该 WA 值超过 MINIMAX 值，不能输出到当前 FO 归并段中，只能等待输出到下一个归并段。

FI

4

6

9

7

13

11

14

22

30

2

3

19

20

WA1

4

4

4

7

7

11

11

22

22

22

3√

3√

3

3

WA2

6

6

6

13

13

13

13

30

30

30

19√

19

19

WA3

9

9

9

9

14

14

14

2√

2√

2√

2

20

MINIMAX

4

6

7

9

11

13

14

22

30

30

2

3

FO

4

6

7

9

11

13

14

22

30

^

2

3

FI

17

1

23

5

36

12

18

21

39

WA1

17

1√

1√

1√

1√

1√

1

18

18

18

WA2

19

19

23

23

36

12√

12

12

12

39

39

39

WA3

20

20

20

5√

5√

5√

5

5

21

21

21

MINIMAX

17

19

20

23

36

36

1

5

12

18

21

39

FO

17

19

20

23

36

^

1

5

12

18

21

39

^

最佳归并树

因为现实中的每个归并段的长度不同，所以归并的次序比较重要。

最佳归并树的衡量

每个初始归并段可以看作一个叶子结点，归并树的长度作为结点权值，则归并树的带权路径长度 WPL 等于读写磁盘的次数。从而归并过程中的磁盘 I/O 次数=归并树的 WPL。

最佳归并树的构造

所以就需要一棵类似哈夫曼树来成为最佳的归并树，不断选择最小的 k 段进行归并。

添加虚段

对于 k 叉归并来说，若初始归并段的数量无法构成严格的 k 叉归并树，则需要补充几个长度为 0 的虚拟段从而能保证严格 k 叉归并，再进行 k 叉哈夫曼树的构造。

那么添加多少虚段呢？

k 叉的最佳归并树一定是一棵严格的 k 叉树，即树中只包含度为 k 和 0 的结点。

设度为 k 的结点有 n_k 个，度为 0 的结点有 n_0 个，归并树的总结点树为 n，则初始归并段数量+虚段数量=n_0。

所以 n=n_0+n_k，kn_k=n-1，所以 n_0=(k-1)n_k+1，所以 n_k=一定是可以整除的。如果不整除就要添加虚段。

形如<!-- -->这类代码无法与 LaTex 代码同时渲染，故此处均删去

# 排序

## 基本概念

- 排序：将一个数据元素的任意序列重新排列成一个按关键字有序的序列。
- 内部排序：待排序的记录存放在计算机的内存中所进行的排序操作称为内部排序。
- 外部排序：待排序的记录数量很大，以致内存一次不能容纳全部记录，在排序过程中需要访问外存的排序过程称为外部排序。
- 稳定的排序：比如一个序列是“1,4,3,3 * ,2”，按从小到大排序后变成“1,2,3,3 * ,4”，就叫做稳定排序，即 3 和 3*相对顺序不变。如果相同关键字的顺序发生了改变，则是不稳定的排序。稳定性的需要看具体的应用场景。
- 内部排序的算法性能取决于算法的时间复杂度和空间复杂度，而时间复杂度一般是由比较和移动次数决定的。外部排序除此之外还要考虑磁盘读写速度和次数。
- 大部分排序算法都仅适用于顺序存储的线性表。
- 大部分排序需要经过比较和移动两个过程。（基数排序不需要比较）排序时至少需要比较$\lceil \log_2(n!) \rceil$次。每次比较两个关键字后，仅出现两种可能的转移。假设整个排序过程至少需要做$t$次比较。则显然会有$2^t$种情况。由于$n$个记录共有$n!$种不同的排列，因而必须有$n!$种不同的比较路径，于是有$2^t \geq n!$，所以得到不等式。

## 插入排序

插入排序的排序序列分为非排序和已排序序列。插入排序就是将选定的目标值插入到对应的位置。

## 直接插入排序

### *直接插入排序过程*

每次将一个待排序的记录按其关键字大小插入到前面已排好序的子序列中，直到全部记录插入完成为止。

### *直接插入排序性能*

空间复杂度为$O(1)$。

时间复杂度主要来自对比关键字，移动元素，若有$n$个元素，则需要$n-1$趟处理。

最好情况是原本的序列就是有序的，需要$n-1$趟处理，每次只需要对比一次关键字，不用移动元素，时间复杂度为$O(n)$。

最坏情况是原本的序列是逆序的，需要$n-1$趟处理，第$i$趟处理需要对比关键字$i+1$次，移动元素$i+2$次，时间复杂度是$O(n^2)$。

所以平均时间复杂度是$O(n^2)$。

直接插入排序算法是稳定的。

如果使用链表实现直接插入排序，移动元素的次数变少了，但是关键字对比次数仍然时$O(n^2)$，从而整体时间复杂度依然是$O(n^2)$。

### *直接插入排序特性*

- 在待排序的元素序列基本有序的前提下，直接插入排序效率最高的。
- 直接插入排序进行$n$躺后能保证前$n+1$个元素是有序的，但是不能保证其都在最终的位置上（如最小的元素在最后一位，这是插入排序的共性）。

## 二分插入排序

### *二分插入排序过程*

也称为折半插入排序，是对直接插入排序的优化，在寻找插入位置时使用二分查找的方式。

当$data[mid] == data[i]$时，为了保证算法的稳定性，会继续在$mid$所指位置右边寻找插入位置。

当$low > high$时停止折半查找，并将$[low, i-1]$内的元素全部右移，并把元素值赋值到$low$所指的位置。

折半插入排序是找到位置了后一起移动元素，而直接插入排序是边查找边排序。

### *二分插入排序性能*

空间复杂度为$O(1)$。

二分插入排序是稳定的。

比起直接插入排序，比较关键字的次数减少为$O(n\log_2 n)$，移动元素的次数没变，所以总体时间复杂度为$O(n^2)$。

*二分插入排序特性*

- 对于直接插入的优化仅在于二分查找的比较次数。
- 二分插入排序进行$n$躺后能保证前$n+1$个元素是有序的，但是不能保证其都在最终的位置上（如最小的元素在最后一位，这是插入排序的共性）。

## 希尔排序

又称缩小增量排序。

*希尔排序过程*

希尔排序也是对直接插入排序的优化。直接插入排序对于基本有序的序列排序效果较好，所以就希望序列能尽可能基本有序。从而希尔排序的思想就是先追求表中元素部分有序，然后逐渐逼近全局有序。

先将整个待排序元素序列分割成若干个子序列（由相隔某个“增量”的元素组成的），分别进行**直接插入排序**，然后缩小增量重复上述过程，直到增量为1。每次对比只对比两个以上的个元素进行插入交换。

增量序列的选择建议是第一趟选择元素个数的一半，后面不断缩小到原来的一半。

*希尔排序性能*

空间复杂度为$O(1)$。

而时间复杂度和增量序列的选择有关，目前无法使用属性手段证明确切的时间复杂度。最坏时间复杂度为$O(n^2)$，在某个范围内可以达到$O(n^{1.3})$。

希尔排序是不稳定的。（因为相同的元素可能分到不同的子序列中进行重排打乱原有顺序）

希尔排序只适用于顺序表而不适合用于链表，无法快速进行增量的访问。

*希尔排序特性*

- 希尔排序在最后一趟前都不能保证元素在最后的位置上。
- 希尔排序在最后一趟前都不能保证元素是有序的。

## 交换排序

交换排序即根据序列中两个元素关键的比较结构然后交换这两个记录在序列中的位置。

## 冒泡排序

重点就是相邻两两比较。

### *冒泡排序过程*

从后往前或从前往后两两比较相邻元素的值，若逆序则交换这两个值，如果相等也不交换，直到序列比较完。这个过程是一趟冒泡排序，第$i$趟后第$i$个元素会已经排序完成。每一趟都会让关键字最小或最大的一个元素到未排序队列的第一个或最后一个。一共需要$n-1$趟排序。

冒泡排序中所产生的有序子序列一定是全局有序的（不同于直接插入排序），也就是说，有序子序列中的所有元素的关键字一定小于或大于无序子序列中所有元素的关键字，这样每趟排序都会将一个元素放置到其最终的位置上。

### *冒泡排序性能*

空间复杂度为$O(1)$。

最好情况下即本身序列有序，则比较次数是$n-1$，交换次数是$0$，从而时间复杂度是$O(n)$。

最坏情况是逆序情况，比较次数和交换次数都是$\frac{n(n-1)}{2}$，所以时间复杂度是$O(n^2)$。

从而平均时间复杂度是$O(n^2)$。

冒泡排序是稳定的。

冒泡排序可以用于链表。

### *冒泡优化*

对于每行冒泡进行优化：如果发现排序前几轮就已经实现了排序成功，那么后面的排序岂不是都浪费了时间进行比较？可以在第一轮循环中设置一个布尔值为$false$，如果在这一轮发生排序交换就设置为$true$，如果一轮结束后发现这个值还是$false$，说明这一轮没有进行交换，表示已经排序成功，就直接所有退出循环。

对于每列冒泡进行优化：默认每一轮冒泡是从$[length-i]$结束，如一共5个元素排序，需要4轮排序，第二轮冒泡排序应该从0开始，到3结束，因为最后一个元素 4 已经在第一轮排序成功。但是如果在第二轮发现2，3已经排序成功了不需要交换，那么默认排序方法第三轮还是要从0到2进行排序，还要比较一次1和2位置的数据，这就造成了浪费，那么如何解决？记录每一轮发生比较的元素的最大索引值，下一轮比较到这个索引值直接结束，不需要继续比较后面的元素。如果最大索引值为0则直接退出。这就进一步优化了上面一种策略。

```
void Bubble(int[] a) {
   int n = a.length - 1;
```

```
    while (true)
    {
        //表示最后一次交换元素位置
        int last = 0;
        for (int i = 0; i < n; i++)
        {
            if (a[i] > a[i + 1]) {
                swap(a, i, i + 1);
                last = i;
            }
        }
        n = last;
        if (n == 0)
        {
            return;
        }
    }
}
```

### *冒泡排序特性*

- 冒泡排序产生的序列全局有序，$n$趟排序后第$n$个元素到达最终的位置上，前$n$个或后$n$个位置的元素确定。

## 快速排序

排序过程类似于构建二叉排序树。基于分治法。

### *快速排序过程*

取待排序序列中的某个元素$pivot$作为基准（一般取第一个元素），通过一趟排序，将待排元素分为左右两个子序列，左子序列元素的关键字均小于或等于基准元素的关键字，右子序列的关键字则大于基准元素的关键字，称进行了一趟快速排序（一次划分），这个$pivot$已经成功排序。然后分别对两个子序列继续进行快速排序，直至整个序列有序。

### *单边循环快排*

即$lomuto$洛穆托分区方案。

1. 选择最右边的元素值做标杆，把标杆值放入$pivot$变量中。
2. 初始时，令$low$和$high$都指向最左边的元素。其中$low$用于被动向右移动，维护小于标杆值的元素的边界，即每次交换的目标索引，一旦交换$low$就向右移动一个；$high$用于主动向右移动，寻找比标杆值小的元素，一旦找到就与$low$指向元素进行交换。
3. 然后$high$开始移动，判断$high$指向的元素值是否小于$pivot$值，如果不小于就继续向右移动。

4. 当遇到比标杆小的值，$high$指向的值就和$low$指向的值进行交换，如果$high$和$low$指向的值为同一个则不进行交换，然后$low$右移一个，$high$继续右移查找。
5. $high$继续移动，最后$high >= pivot$时将基准点元素值与$low$指向值进行交换，该轮排序结束。此时$low$指向的位置就是$pivot$值所应该在的位置。
6. 返回基准点元素所在索引，从而确定排序上下边界，递归继续执行排序。

*双边循环快排*

分为普通分区方案和$hoare$霍尔分区方案。逻辑基本上一样，只是边界选择方式不同。

1. 先选择个值做标杆，一般为最左边值，把标杆值放入$pivot$变量中。
2. 初始时，令$high$指向序列最右边的值，$low$指向序列最左边的值。
3. 然后从$high$开始不断左移，当遇到比标杆大或等于的值时$high--$。
4. 如果发现比标杆小的值，即$high < pivot$，需要交换，然后$high$不动，$low++$开始移动去找要交换的大于标杆的值。
5. 若$low$所指向的值比标杆小或等于，则$low++$进行寻找。（为什么要加上一个等于标杆值可以继续向右移动的条件？因为标杆值默认是第一个值，即初始化$pivot$跟$low$指向同一个值，如果只能小于标杆值才能继续移动不交换，则在第一个元素时，由于标杆值被初始化赋值为第一个值，则标杆$pivot$值等于$low$值，从而导致$low$值跟$high$值进行交换，导致$pivot$标杆值被交换走了，标杆值变为了最开始最右边的$high$值，导致了排序有问题）
6. 若$low$所指向的值比标杆大，则$low$值与$high$值进行交换。
7. 交换后$low$不动，$high--$开始移动。回到步骤三开始执行。
8. 当$low = high$时表示$low$和$high$之前的元素都比基准小，$low$和$high$之后的元素都比基准大，完成了一次划分。然后把基准元素放入$low == high$指向的位置。
9. 不断交替使用$low$和$high$指针进行对比。对左右子序列进行同样的递归操作即可，从步骤三开始。若左右两个子序列的元素数量小等于一，则无需再划分。

即对序列进行比较，有头尾两个指针，尾指针开始比较向前移动，若指向值比对比值小则要交换，交替让头指针开始移动，否则不改变指针则尾指针继续向前；同理头指针向后移动，若指向值比对比值大则交换，交替让尾指针移动，否则不改变指针则头指针继续向后。最后头尾指针指向一个位置，将对比值插入到当前值，此时一趟完成。

洛穆托分区与霍尔分区比较。

*快速排序性能*

由于快速排序使用了递归，所以需要递归工作栈，空间复杂度与递归层数相关，所以为$O$(递归层数)。

每一层划分只需要处理剩余的待排序元素，时间复杂度不超过$O(n)$，所以时间复杂度为$O(n \times$归层数)。

而快速排序会将所有元素组织成为二叉树，二叉树的层数就是递归调用的层数。所以对于$n$个结点的二叉树，最小高度为$\lfloor \log_2 n \rfloor + 1$，最大高度为$n$。

从而最好时间复杂度为$O(n\log_2 n)$，最坏时间复杂度为$O(n^2)$，平均时间复杂度为$O(n\log_2 n)$；最好空间复杂度为$O(\log_2 n)$，最坏空间复杂度为$O(n)$，平均空间复杂度为$O(\log_2 n)$。

所以如果初始序列是有序的或逆序的，则快速排序性能最差（速度最慢）。若每一次选中的基准能均匀划分，尽量让数轴元素平分，则效率最高（速度最快）。性能与分区处理顺序无关。

所以对于快速排序性能优化是选择尽可能能中分的基准元素，入选头中尾三个位置的元素，选择中间值作为基准元素，或随机选择一个元素作为基准元素。

最好使用顺序存储，这样找到数轴元素与遍历时比较简单。

快速排序算法是不稳定的。

*快速排序特性*

- 快速排序不产生有序子序列。
- 枢轴元素到达的位置是不确定的，但是每次都会到其最终的位置上。第$n$趟有$n$个元素到最终位置上。
- 求快速排序趟数就是找到符合这种性质的元素个数。
- 快速排序在内部排序中的表现最好。
- 对于基本有序或倒序的序列，快速排序速度最慢。
- 对于每次的数轴元素能尽量将表分为长度相同的子表，快速排序速度最快。

## 选择排序

分为已排序和未排序序列。选择排序就是每一趟在待排序元素中选取关键字最小或最大的元素加入有序子序列。

选择排序也需要交换，但是与交换排序的不断交换不同的是选择排序时选择出一个最后进行交换，一趟只交换一次。

选择排序也需要插入，且也分为已排序和未排序序列，但是插入排序不需要选择，且元素移动方式是插入而不是交换。

选择排序算法的比较次数始终为$n(n-1)/2$，与序列状态无关。

## 简单选择排序

### *简单选择排序过程*

即每一趟在待排序元素中选取关键字最小的元素加入有序序列。交换发生在选出最值后，在每趟的尾部。经过$n-1$趟就可以完成。

### *简单选择排序性能*

空间复杂度为$O(1)$。

时间复杂度为$O(n^2)$。

简单选择排序是不稳定的。因为选择后会进行交换，影响顺序。

简单选择排序也可以适用于链表。

### *直接插入排序与简单选择排序*

插入排序和选择排序都是分为未排序和已排序两个部分，那么其中有什么区别？

如18、23、19、9、23 *、15进行排序。

18 23 19 9 23* 15

插入排序：

```
18 23 19 9 23* 15
18 19 23 9 23* 15
9 18 19 23 23* 15
9 18 19 23 23* 15
9 15 18 19 23 23*
```

选择排序：

```
9 23 19 18 23* 15
9 15 19 18 23* 23
9 15 18 19 23* 23
9 15 18 19 23* 23
9 15 18 19 23* 23
9 15 18 19 23* 23
```

## 堆排序

### *堆的定义*

若$n$个关键字序列$L$满足下面某一条性质，则就是堆：

1. 若满足$L(i) \geq L(2i)$且$L(i) \geq L(2i+1)$ $\left(1 \leq i \leq \frac{n}{2}\right)$则是大根堆或大顶堆。

2. 若满足$L(i) \leq L(2i)$且$L(i) \leq L(2i+1)$ $\left(1 \leq i \leq \frac{n}{2}\right)$则是小根堆或小顶堆。

所以堆就是用顺序存储的完全二叉树。

堆的叶子结点范围是$\lfloor \log_2 n \rfloor + 1 \sim n$。

*堆的建立*

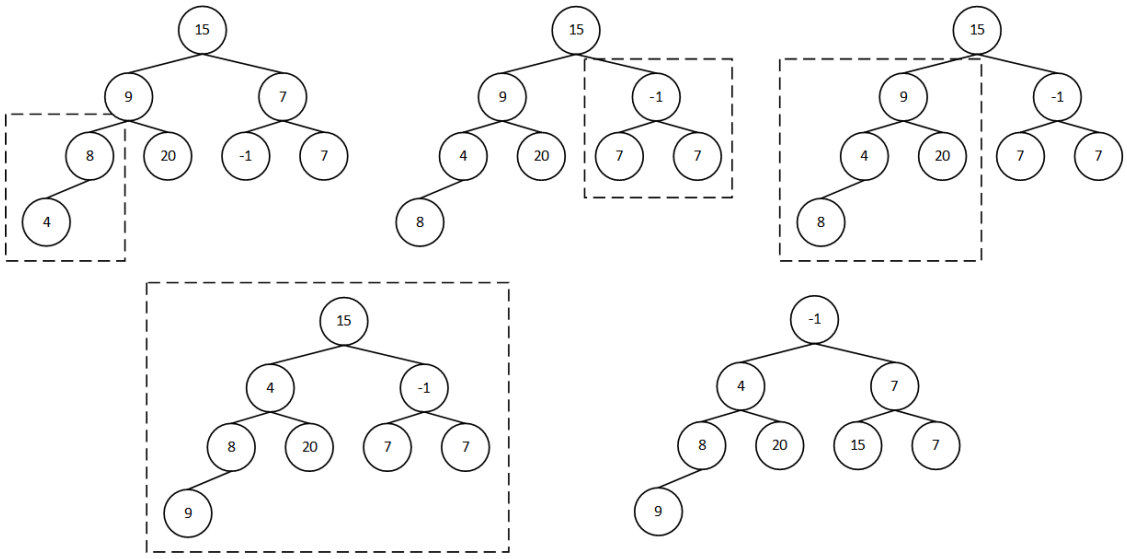
其实堆就是层序存储的完全二叉树。其中：

- $i \leq \lfloor \frac{n}{2} \rfloor$的结点都是非终端结点。
- $i$的左孩子是$2i$。
- $i$的右孩子是$2i+1$。
- $i$的父结点是$\lfloor \frac{n}{2} \rfloor$。

所以建立根堆过程是：

1. 按照关键字序列依次添加关键字到二叉树，按照层次遍历顺序添加。
2. 初始化成功后再从下往上、从左至右按逆层次遍历顺序不断调整位置。
3. 如果是大根堆则大元素往上，且当前结点与更大的孩子结点互换；如果是小根堆则小元素往上，且当前结点与更小的孩子结点互换。
4. 递归往上时父子结点不断交换位置。
5. 如果元素互换破坏了调整好的下一级的堆，则使用同样的方法对下一层递归调整。

如用堆排序对$(15,9,7,8,20,-1,7,4)$建立小根堆堆。首先将这组数据按层序初始化为无序堆，然后从最后向前开始调整：




*建立堆*

1. 从$t < \lfloor \frac{n}{2} \rfloor$的结点开始往前遍历。
2. 检查当前结点$i$与左孩子和右孩子是否满足根堆条件，若不满足则交换。
   - 若是建立大根堆，检查是否满足根大于等于左、右结点，若不满足，则当前结点与更大的一个孩子互换。
   - 若是建立小根堆，检查是否满足根小于等于左、右结点，若不满足，则当前结点与更小的一个孩子互换。
3. 若元素互换破坏了下一级的堆，则采用同样的方法继续向下调整。
   - 若是建立大根堆，则小的元素不断下坠。
   - 若是建立小根堆，则大的元素不断下坠。 ->

调整堆的时间与树高相关$O(\log_2 n)$，建立堆的时间复杂度为$O(n)$。比较总次数不超过$4n$。

*堆排序过程*

由于选择排序是在每一趟都选择最大或最小的值进行排序，所以堆排序中就通过堆这个存储结构来完成对最值的选取——直接选择堆顶元素。

堆排序即每次将堆顶元素与堆底元素（堆最底层最右元素）进行交换，表示这个部分已经排序完成了不需要进行调整，第$i$趟表示倒数$i$个元素已经有序，所以无序的元素就是堆前面的元素。

1. 每一趟将堆顶元素加入子序列（堆顶元素与待排序序列中的最后一个元素交换）。此时后面的这个元素就排序好了。最右下的元素作为堆顶元素。
2. 此时待排序序列已经不是堆了（堆顶不能保证是最小或最大的元素），需要将其再次调整为堆（小元素或大元素不断下坠）。
3. 重复步骤一二。
4. 直到$n-1$趟处理后得到有序序列。基于大根堆的堆排序会得到递增序列，而基于小根堆的堆排序会得到递减序列。

调整堆从右边即序列末尾开始。

*堆排序性能*

堆排序的存储就是它本身，不需要额外的存储空间，要么只需要一个用于交换或临时存放元素的辅助空间。所以空间复杂度为$O(1)$。

若树高为$h$，某结点在第$i$层，则将这个结点向下调整最多只需要下坠$h-i$层，关键字对比次数不超过$2(h-i)$次。

第$i$层最多$2^{i-1}$个结点，而只有第$1 \cdots (h-1)$层的结点才可能需要下坠调整。所以调整时关键字对比次数不超过$\sum_{i=h-1}^{12} {}^{i-1}\, 2(h-i) = \sum_{j=1}^{h-1} 2^{h-j} j \leq 2n \sum_{j=1}^{h-1} \frac{j}{2^j} \leq 4n$。

所以建堆过程中，关键字对比次数不超过$4n$，建堆的时间复杂度为$O(n)$。

堆排序中处理时根结点最多下坠$h-1$层，而每下坠一层，最多对比关键字两次，所以每一趟排序的时间复杂度不超过$O(h)=O(\log_2 n)$，一共$n-1$趟，所以时间复杂度为$O(n\log_2 n)$。所以总的时间复杂度也是$O(n\log_2 n)$。

堆排序是不稳定的。

堆排序适合关键字较多的情况。创如，在1亿个数中选出前100个最大值，首先使用一个大小为100的数组，读入前100个数，建立小顶堆，而后依次读入余下的教，若小于堆顶则舍弃，否则用该数取代堆顶并重新调整堆，待数据读取完毕，堆中100个数即为所求。

*堆的插入*

新元素放到表尾（即最右下角元素），并与其$\lfloor\frac{i}{2}\rfloor$的父结点进行对比，若新元素比父元素更大（大根堆）或更小（小根堆），则二者互换，并保持上升，直到无法上升为止。时间复杂度为树高$O(\log_2 n)$。

*堆的删除*

被删除的元素用堆底元素（即最右下角元素）代替，然后让这个元素不断下坠，直到无法下坠为止。时间复杂度为树高$O(\log_2 n)$。

*堆排序特性*

- 适合大量数据进行排序。
- 在含有$n$个关键字的小根堆中，关键字最大的记录存储范围为$\lfloor\frac{n}{2}\rfloor+1\sim n$。这是小根堆，关键字最大的记录一定存储在这个堆所对应的完全二叉树的叶子结点中；又因为二叉树中的最后一个非叶子结点存储在$\lfloor\frac{n}{2}\rfloor$中，所以得到范围。

## 归并排序

归并是指把两个（二路归并）或多个（多路归并）已经有序的序列合并为一个。

该算法是采用分治法的一个非常典型的应用。将已有序的子序列合并，得到完全有序的序列；即先使每个子序列有序，再使子序列段间有序。

在较大数据进行排序时为了加快速度使用归并排序，用空间换时间。

### 二路归并排序

二路归并排序比较常用，且基本上用于内部排序，多路排序多用于外部排序。

*二路归并排序过程*

1. 把长度为$n$的输入序列分成两个长度为$\frac{n}{2}$的子序列。

2. 对这两个子序列分别采用归并排序。
3. 将两个排序好的子序列合并成一个最终的排序序列。

归并排序趟数为$\lceil \log_2 n \rceil$。

*二路归并排序性能*

二路归并排序是一棵倒立的二叉树。

空间复杂度主要来自辅助数组，所以为$O(n)$，而递归调用的调用栈的空间复杂度为$O(\log_2 n)$，总的空间复杂度就是为$O(n)$，无论平均还是最坏，所以这个算法在内部排序算法中空间消耗最大。

$n$个元素二路归并排序，归并一共要$\log_2 n$趟，每次归并时间复杂度为$O(n)$，则算法时间复杂度为$O(n\log_2 n)$

归并排序是稳定的。

## 分配排序

分配排序过程无须比较关键字，而是通过用额外的空间来“分配”和“收集”来实现排序，它们的时间复杂度可达到线性阶$O(n)$。简言之就是：用空间换时间，所以性能与基于比较的排序才有数量级的提高。

### 基数排序

基数排序不是基于比较的排序算法，其核心在于将输入的数据值转化为键存储在额外开辟的数组空间中。作为一种线性时间复杂度的排序，计数排序要求输入的数据必须是有确定范围的整数。

- 只能对整数进行排序。
- 元素的移动次数与关键字的初始排列次序无关。

*基数的定义*

假设长度为$n$的线性表中每个结点$a_j$的关键字由$d$元组$\left(k_j^{d-1}, k_j^{d-2}, \cdots, k_j^1, k_j^0\right)$组成，其中$0 \geq k_j^i \geq r-1\ (0 \geq i \geq d-1)$，其中$r$就是基数。

*基数排序过程*

有最高位优先$MSD$和最低位优先$LSD$两种方法。

若是要得到递减序列：

1. 初始化：设置$r$个空辅助队列$Q_{r-1}, Q_{r-2}, \cdots, Q_0$。
2. 按照每个关键字位**权重递增**的次序（个、十、百），对$d$个关键字位分别做分配和收集。

3. 分配就是顺序扫描各个元素，若当前处理的关键字位为$x$，就将元素插入$Q_x$队尾。
4. 收集就是把$Q_{r-1}, Q_{r-2}, \cdots, Q_0$各个队列的结点依次出队并链接在一起。

### *基数排序性能*

基数排序基本上使用链式存储而不是一般的顺序存储。

需要$r$个辅助队列，所以空间复杂度为$O(r)$。

一趟分配$O(n)$，一趟收集$O(r)$，一共有$d$趟分配收集，所以总的时间复杂度为$O\big(d(n+r)\big)$。与序列初始状态无关。

基数排序是稳定的。

### *基数排序的应用*

对于一般的整数排序是可以按位排序的，也可以处理一些实际问题，如根据人的年龄排序，需要从年月日三个维度分别设置年份的队列、月份的队列（1到12）、日的队列（1到31）。

所以基数排序擅长解决的问题：

1. 数据元素的关键字可以方便地拆分为$d$组，且$d$较小。
2. 每组关键字的取值范围不大，即$r$较小。
3. 数据元素个数$n$较大。

## 计数排序

作为一种线性时间复杂度的排序，计数排序要求输入的数据必须是有确定范围的整数。

### *计数排序过程*

1. 找出待排序的数组中最大和最小的元素。
2. 统计数组中每个值为 i 的元素出现的次数，存入数组 C 的第 i 项。
3. 对所有的计数累加（从 C 中的第一个元素开始，每一项和前一项相加）。
4. 反向填充目标数组：将每个元素 i 放在新数组的第 C(i)项，每放一个元素就将 C(i)减去 1。

当输入的元素是$n$个属于$[0, k]$的整数时，时间复杂度是$O(n+k)$，空间复杂度也是$O(n+k)$，其排序速度快于任何比较排序算法。

当$k$不是很大并且序列比较集中时，计数排序是一个很有效的排序算法。

计数排序是稳定的。

### 桶排序

桶排序是计数排序的升级版。它利用了函数的映射关系，高效与否的关键就在于这个映射函数的确定。桶排序的工作的原理：假设输入数据服从均匀分布，将数据分到有限数量的桶里，每个桶再分别排序（有可能再使用别的排序算法或是以递归方式继续使用桶排序进行排）。

#### *桶排序过程*

1. 设置一个定量的数组当作空桶。
2. 遍历输入数据，并且把数据一个一个放到对应的桶里去。
3. 对每个不是空的桶进行排序。
4. 从不是空的桶里把排好序的数据拼接起来。

#### *桶排序性能*

桶排序最好情况下使用线性时间$O(n)$，桶排序的时间复杂度，取决与对各个桶之间数据进行排序的时间复杂度，因为其它部分的时间复杂度都为$O(n)$。桶排序的平均时间复杂度为线性的$O(n+C)$，其中$C=n\times(\log n-\log m)$，其中$m$代表桶划分的数量。

很显然，桶划分的越小，各个桶之间的数据越少，排序所用的时间也会越少。但相应的空间消耗就会增大。

桶排序是稳定的。

## 内部排序

指在排序期间元素全部存放在内存中的排序。除了分配排序，其他的内部排序往往要经过比较和移动。

| 算法种类 | 最好时间复杂度 | 平均时间复杂度 | 最好时间复杂度 | 空间复杂度 | 是否稳定 | 趟数 |
|---|---|---|---|---|---|---|
| 直接插入排序 | $O(n)$ | $O(n^2)$ | $O(n^2)$ | $O(1)$ | 是 | $n-1$ |
| 希尔排序 | ? | ? | ? | $O(1)$ | 否 | $s$ |
| 简单选择排序 | $O(n^2)$ | $O(n^2)$ | $O(n^2)$ | $O(1)$ | 否 | $n-1$ |
| 快速排序 | $O(n\log_2 n)$ | $O(n\log_2 n)$ | $O(n^2)$ | $O(\log_2 n)$ | 否 | 初始序列 |
| 冒泡排序 | $O(n)$ | $O(n^2)$ | $O(n^2)$ | $O(1)$ | 是 | 初始序列 |

| 算法种类 | 最好时间复杂度 | 平均时间复杂度 | 最好时间复杂度 | 空间复杂度 | 是否稳定 | 趟数 |
|---|---|---|---|---|---|---|
| 堆排序 | $O(n\log_2 n)$ | $O(n\log_2 n)$ | $O(n\log_2 n)$ | $O(1)$ | 否 | 初始序列 |
| 二路归并排序 | $O(n\log_2 n)$ | $O(n\log_2 n)$ | $O(n\log_2 n)$ | $O(n)$ | 是 | $\log_2 n$ |
| 基数排序 | $O(d(n+r))$ | $O(d(n+r))$ | $O(d(n+r))$ | $O(r)$ | 是 | $r$ |

- 每趟排序结束都至少能够确定一个元素最终位置的方法：选择、交换。（插入和归并则不行）

## 外部排序

指在排序期间元素无法全部同时存放在内存中，必须在排序过程中根据要求不断地在内、外存之间移动的排序。

### 外部排序的原理

#### *外部排序过程*

磁盘的读写是以块为单位，数据读入内存后才能被修改，修改完成后还需要写回磁盘。

外部排序就是针对数据元素太多，无法一次性全部读入内存进行排序而进行处理的在外部磁盘进行的排序处理方式。

使用归并排序的方式，最少只用在内存分配三块大小的缓冲区（两个输入缓冲一个输出缓冲）即可堆任意一个大文件进行排序。然后对缓冲区里的数据进行内部排序。

外部排序过程：

1. 生成初始归并段（大小为输入缓冲区的总大小），需要读写并进行内部排序。
2. 重复读写，进行内部归并排序。填满输出缓冲就可以输出。输入缓冲空就可以输入新数据。

外部排序时间开销=读写外存时间（最大的时间开销）+内部排序所需时间+内部归并所需时间。

#### *外部排序的优化方法*

优化方法就是使用更多路的多路归并，减少归并趟数。

$k$路平衡归并：最多只能有$k$个段归并为一个，需要一个输出缓冲区和$k$个输入缓冲区；每一趟归并中，若有$m$个归并段参与归并，则经过这一趟处理得到$\lceil \frac{m}{k} \rceil$个新的归并段。

对$r$个初始归并段，使用$k$路归并，则归并树可以使用$k$叉树表示，若树高为$h$，则归并趟数最小为$h = \lceil \log_k r \rceil + 1$。

但是多路归并会带来负面影响：

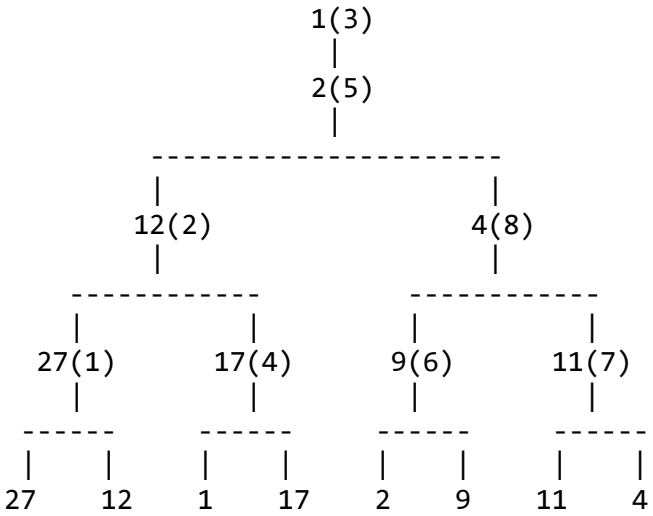
1. $k$路归并时，需要开辟$k$个输入缓冲区，内存开销增加。
2. 每挑选一个关键字需要对比关键字$(k-1)$次，内部归并时间增加。

同时，若能增加初始归并段的长度$k$，也可以减少初始归并段数量$r$从而进行优化。

### 败者树

用于通过过去归并的经历减少归并次数。败者树可以看作一棵多了一个单个的根的完全二叉树。$k$个叶结点分别是当前参加比较的元素，非叶子结点用来记忆左右子树中的失败者，而让胜者往上继续比较，一直到根结点。

如要用败者树排序27,12,1,17,2,9,11,4，格式：元素值(归并段索引值)：




传统方法从$k$个归并段选出一个最大或最小元素需要对比关键字$k-1$次，而使用$k$路归并的败者树只需要对比关键字$\lceil \log_2 k \rceil$次（败者树层数，不包括成功结点）。

构建败者树还是需要$n-1$次对比。

### 置换选择排序

如果内存工作区只能容纳$l$个记录，则初始归并段也只能包含$l$条记录，若文件共有$n$条记录，则初始归并段的数量为$r = \lceil n/l \rceil$。

用于构建更长的初始归并段，从而减少归并次数。

假设初始始待排文件为$FI$，初始归并段输出文件为$FO$，内存工作区为$WA$，$FO$和$WA$的初始状态为空，$WA$可容纳$w$个记录。置换选择算法的步骤如下：

1. 从$FI$输入$w$记录到工作区$WA$。
2. 从$WA$中选出其中关键字取最小值的记录，记为$MINIMAX$记录。
3. 将$MINIMAX$记录输出到$FO$中去。
4. 若$FI$不空，则从$FI$输入下一个记录到$WA$中。
5. 从$WA$中所有关键字比$MINIMAX$记录的关键字大的记录中选出最小关键字记录，作为新的$MINIMAX$记录。
6. 重复步骤三到五，如果新输入到$FI$的关键字小于$MINIMAX$的值，则驻留在$WA$中，直至在$WA$中填满选不出新的$MINIMAX$记录为止，由此得到一个初始归并段，输出一个归并段的结束标志到$FO$中去。准备输出新的归并段。
7. 重复步骤二到六，直至$WA$为空。由此得到全部初始归并段。

此时输出的初始归并段可以超过$WA$，且初始归并段长度是不一定相等的。

如$FI$：4,6,9,7,13,11,14,22,30,2,3,19,20,17,1,23,5,36,12,18,21,39，$WA$长度为3，$FO$为4,6,7,9,11,13,14,16,22,30、2,3,10,17,19,20,23,36、1,5,12,18,21,39。

^表示中断当前归并段，下一个$FI$开启新的归并段；√表示该$WA$值超过$MINIMAX$值，不能输出到当前$FO$归并段中，只能等待输出到下一个归并段。

| FI | 4 | 6 | 9 | 7 | 13 | 11 | 14 | 22 | 30 | 2 | 3 | 19 | | 20 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| WA1 | 4 | 4 | 4 | 7 | 7 | 11 | 11 | 22 | 22 | 22 | 3√ | 3√ | 3 | 3 |
| WA2 | | 6 | 6 | 6 | 13 | 13 | 13 | 13 | 30 | 30 | 30 | 19√ | 19 | 19 |
| WA3 | | | 9 | 9 | 9 | 9 | 14 | 14 | 14 | 2√ | 2√ | 2√ | 2 | 20 |
| MINIMAX | | | 4 | 6 | 7 | 9 | 11 | 13 | 14 | 22 | 30 | 30 | 2 | 3 |
| FO | | | 4 | 6 | 7 | 9 | 11 | 13 | 14 | 22 | 30 | ^ | 2 | 3 |

| FI | 17 | 1 | 23 | 5 | 36 | 12 | | 18 | 21 | 39 | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| WA1 | 17 | 1√ | 1√ | 1√ | 1√ | 1√ | 1 | 18 | 18 | 18 | | | |
| WA2 | 19 | 19 | 23 | 23 | 36 | 12√ | 12 | 12 | 12 | 39 | 39 | 39 | |
| WA3 | 20 | 20 | 20 | 5√ | 5√ | 5√ | 5 | 5 | 21 | 21 | 21 | | |
| MINIMAX | 17 | 19 | 20 | 23 | 36 | 36 | 1 | 5 | 12 | 18 | 21 | 39 | |
| FO | 17 | 19 | 20 | 23 | 36 | ^ | 1 | 5 | 12 | 18 | 21 | 39 | ^ |

## 最佳归并树

因为现实中的每个归并段的长度不同，所以归并的次序比较重要。

### *最佳归并树的衡量*

每个初始归并段可以看作一个叶子结点，归并树的长度作为结点权值，则归并树的带权路径长度$WPL$等于读写磁盘的次数。从而归并过程中的磁盘$I/O$次数=归并树的$WPL \times 2$。

*最佳归并树的构造*

所以就需要一棵类似哈夫曼树来成为最佳的归并树，不断选择最小的$k$段进行归并。

*添加虚段*

对于$k$叉归并来说，若初始归并段的数量无法构成严格的$k$叉归并树，则需要补充几个长度为0的虚拟段从而能保证严格$k$叉归并，再进行$k$叉哈夫曼树的构造。

那么添加多少虚段呢？

$k$叉的最佳归并树一定是一棵严格的$k$叉树，即树中只包含度为$k$和0的结点。

设度为$k$的结点有$n_k$个，度为0的结点有$n_0$个，归并树的总结点树为$n$，则初始归并段数量+虚段数量=$n_0$。

所以$n = n_0 + n_k$，$kn_k = n - 1$，所以$n_0 = (k-1)n_k + 1$，所以$n_k = \frac{(n_0-1)}{(k-1)}$一定是可以整除的。如果不整除就要添加虚段。