数据结构

一. 什么是数据结构

1. 决定了数据的顺序和位置关系

数据存储于计算机的内存中。内存如下图所示，形似排成 1 列的箱子，1 个箱子里存储 1 个数据。

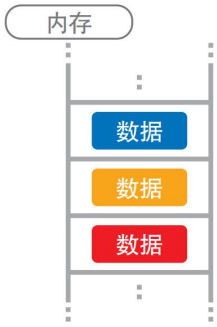


图 1. 数据存储于内存中

数据存储于内存时，决定了数据顺序和位置关系的便是“数据结构”。

2. 电话簿的数据结构

1) 从上往下顺序添加

举个简单的例子。假设我们有 1 个电话簿——虽说现在很多人都把电话号码存在手机里，但是这里我们考虑使用纸质电话簿的情况——每当我们得到了新的电话号码，就按从上往下的顺序把它们记在电话簿上。

姓名	电话号码
小小	010-xxxx-xxxx
大大	010-xxxx-xxxx
刘慧玲	010-xxxx-xxxx
李廷钰	010-xxxx-xxxx
。。。。。。	。。。。。。

假设此时我们想给“刘慧玲”打电话，但是因为数据都是按获取顺序排列的，所以我们并不知道刘慧玲的号码具体在哪里，只能从头一个个往下找（虽说也可以“从后往前找”或者“随机查找”，但是效率并不会比“从上往下找”高）。如果电话簿上号码不多的话很快就能找到，但如果存了 500 个号码，找起来就不那么容易了。

2) 按姓名的拼音顺序排列

接下来，试试以联系人姓名的拼音顺序排列。因为数据都是以字典顺序排列的，所以它们是有“结构”的。

姓名	电话号码
刘慧玲	010-xxxx-xxxx
李廷钰	010-xxxx-xxxx
毛毛	010-xxxx-xxxx
钟宁暄	010-xxxx-xxxx
。。。。。。	。。。。。。

使用这种方式给联系人排序的话，想要找到目标人物就轻松多了。通过姓名的拼音首字

母就能推测出该数据的大致位置。

那么，如何往这个按拼音顺序排列的电话簿里添加数据呢？假设我们认识了新朋友“奇衡三”并拿到了他的电话号码，打算把号码记到电话簿中。由于数据按姓名的拼音顺序排列，所以奇衡三必须写在毛毛和钟宁暄之间，但是上面的这张表里已经没有空位可供填写，所以需要把钟宁暄及其以下的数据往下移 1 行。

此时我们需要从下往上执行“将本行的内容写进下一行，然后清除本行内容”的操作。如果一共有 500 个数据，一次操作需要 10 秒，那么 1 个小时也完成不了这项工作。

3) 两种方法的优缺点

总的来说，数据按获取顺序排列的话，虽然添加数据非常简单，只需要把数据加在最后就可以了，但是在查询时较为麻烦；以拼音顺序来排列的话，虽然在查询上较为简单，但是添加数据时又会比较麻烦。

虽说这两种方法各有各的优缺点，但具体选择哪种还是要取决于这个电话簿的用法。如果电话簿做好之后就不再添加新号码，那么选择后者更为合适；如果需要经常添加新号码，但不怎么需要再查询，就应该选择前者。

4) 将获取顺序与拼音顺序结合

我们还可以考虑一种新的排列方法，将二者的优点结合起来。

那就是分别使用不同的表存储不同的拼音首字母，比如表 L、表 X、表 Z 等，然后将同一张表中的数据按获取顺序进行排列。

a. 表 L

姓名	电话号码
刘慧玲	010-xxxx-xxxx
李廷钰	010-xxxx-xxxx
刘彻	010-xxxx-xxxx
刘静	010-xxxx-xxxx
。。。。。。	。。。。。。

b. 表 X

姓名	电话号码
肖凯文	010-xxxx-xxxx
肖晓华	010-xxxx-xxxx
。。。。。。	。。。。。。
。。。。。。	。。。。。。
。。。。。。	。。。。。。

c. 表 Z

姓名	电话号码
曾铖	010-xxxx-xxxx
曾革建	010-xxxx-xxxx

曾翠梅	。。。。。。
。。。。。。	。。。。。。
。。。。。。	。。。。。。

这样一来，在添加新数据时，直接将数据加入到相应表中的末尾就可以了，而查询数据时，也只需要到其对应的表中去查找即可。

因为各个表中存储的数据依旧是没有规律的，所以查询时仍需从表头开始找起，但比查询整个电话簿来说还是要轻松多了。

3. 选择合适的数据结构以提高内存的利用率

数据结构方面的思路也和制作电话簿时的一样。将数据存储于内存时，根据使用目的选择合适的数据结构，可以提高内存的利用率。

4. 常见数据结构时间复杂度

数据结构	插入	访问	查找	删除	备注
Array	O(n)	O(1)	O(n)	O(n)	插入最后位置的复杂度为 O(1)
HashMap	O(1)	O(1)	O(1)	O(1)	重新计算哈希会影响插入时间
Map	O(log(n))		O(log(n))	O(log(n))	通过二叉搜索树实现
Stack	O(1)			O(1)	插入与删除都遵循后入先出(LIFO)
Queue（array 实现）	O(n)			O(1)	
Queue（List 实现）	O(1)			O(1)	使用双向链表
Linked List（单向）	O(n)		O(n)	O(n)	在起始位置添加或删除元素，复杂度为 O(1)
Linked List（双向）	O(n)		O(n)	O(n)	在起始或结尾添加或删除元素，复杂度为 O(1)。然而在其他位置是 O(n)。

二. 绪论

1. 基本概率和术语

1) **数据**(Data)

是信息的载体，是描述客观事物属性的数、字符及所有能输入到计算机中并被计算机程序识别和处理的符号的集合。

对客观事物的符号表示，在计算机科学中是指所有能输入到计算机中并被计算机程序处理的符号的总称。

2) **数据元素**(Data Element)

是数据的基本单位。有时，一个数据元素可由若干个数据项(Data Item)组成。例如：一本书的书目信息为一个数据元素，而书目信息中的每一项（如书名、作者名等）为一个数据项。数据项是数据的不可分割的最小单位。

3) **数据对象**(Data Object)

是性质相同的数据元素的集合，是数据的一个子集。

4) **抽象数据类型**（ADT）

描述了数据的逻辑结构和抽象运算，通常用（数据对象，数据关系，基本操作集）这样的三元组表示，从而构成一个完整的数据结构

5) **数据结构**(Data Structure)

是相互之间存在一种或多种特定关系的数据元素的集合，包括逻辑结构、存储结构和数据运算。

2. 数据结构三要素

逻辑结构、存储结构（物理结构）、数据的运算。

1) 逻辑结构

是指数据元素之间的逻辑关系，与数据的存储无关，是独立于计算机的分类。分为线性结构（线性表、栈、队列）和非线性结构（树、图、集合）。

2) 存储结构

是用计算机语言实现的逻辑结构，它依赖于计算机语言。

a. 顺序存储

① 优点是可以实现随机存取，每个元素占用最少的存储空间；
② 缺点是只能使用相邻的一块存储单元，因此可能产生较多的外部碎片。

b. 链式存储

① 优点是不会出现碎片现象，能充分利用所有存储单元；
② 缺点是每个元素因存储指针而占用额外的存储空间，且只能实现顺序存取。

c. 索引存储

① 优点是检索速度快；
② 缺点是增加附加的索引表后会占用较多的存储空间。另外，在增加和删除数据时要修改索引表，因而会花费较多的时间。

d. 散列存储

① 优点是检索、增加和删除结点的操作都很快；
② 缺点是若散列函数不好，则可能出现元素存储单元冲突，而解决冲突会增加时间和空间开销。

3. 运算

定义是针对逻辑结构的，指出运算的功能；实现是针对存储结构的，指出运算的具体操作步骤。

4. 小结

① 循环队列是用顺序表表示的队列，是一种数据结构。
② 栈是一种抽象数据类型，可采用顺序存储或链式存储，只表示逻辑结构。
③ 线性表、栈和队列的逻辑结构都是相同的，都属于线性结构，只是他们对数据的运算不同从而表现出不同的特点。

三. 算法和算法评价

1. 算法的基本概念

a. 算法：

是对特定问题求解步骤的一种描述，它是指令的有限序列，其中的每一条指令都表示一个或多个操作。

b. 算法的特性：

① 有穷性

一个算法（对任何合法的输入值）必须总是在执行有穷步之后结束，且每一步都可在有穷时间内完成。

② 确定性

对于相同的输入只能得出相同的输出。

③ 可行性

算法中所有描述的操作都是可以通过已经实现的基本运算执行有限次来实现的。

④ 输入

一个算法有零个或多个输入，这些输入取自于某个特定的对象集合。

⑤ 输出

一个算法有一个或多个输出，这些输出是与输入有着某种特定关系的量。

c. 算法的设计目标：

正确性、可读性、健壮性、效率与低存储需求。

2. 算法效率的度量

a. 时间复杂度

频度指该语句在算法中被重复执行的次数。

算法的时间复杂度不仅依赖于问题的规模 n，也取决于输入数据的性质。

b. 空间复杂度

算法原地工作是指算法所需的辅助空间为常量，即 O（1）。

3. 总结

同一个算法，实现语言的级别越高，执行效率越低。

在已经确定时间复杂度的情况下比较算法，不用再考虑特殊输入值。

四. 链表

链表是数据结构之一，其中的数据呈线性排列。在链表中，数据的添加和删除都较为方便，就是访问比较耗费时间。

1. 单链表

图 2.　　单链表

这就是链表的概念图。Blue、Yellow、Red 这 3 个字符串作为数据被存储于链表中。每个数据都有 1 个“指针”，它指向下一个数据的内存地址。

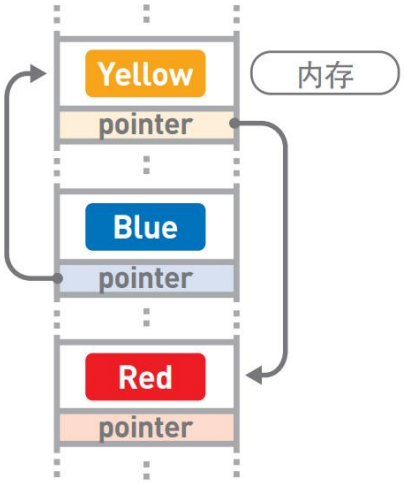


图 3.　　链表的内存空间不连续

在链表中，数据一般都是分散存储于内存中的，无须存储在连续空间内。

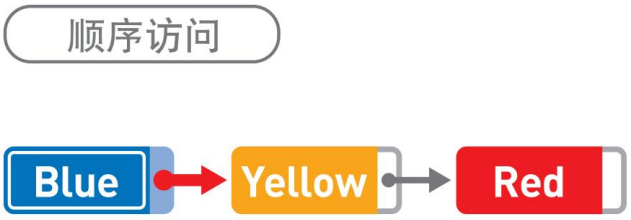


图 4.　　单链表只能顺序访问

因为数据都是分散存储的，所以如果想要访问数据，只能从第 1 个数据开始，顺着指针的指向一一往下访问（这便是顺序访问）。比如，想要找到 Red 这一数据，就得从 Blue 开始访问。

这之后，还要经过 Yellow，我们才能找到 Red。

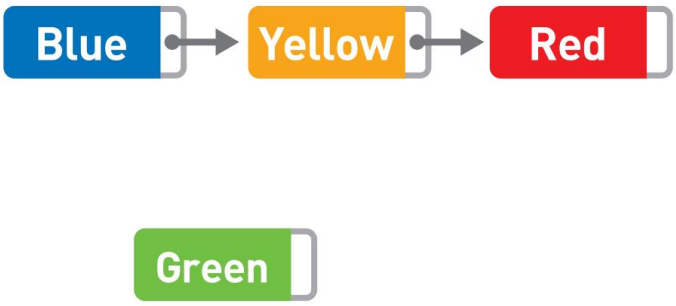


图 5.　　单链表插入数据

如果想要添加数据，只需要改变添加位置前后的指针指向就可以，非常简单。比如，在 Blue 和 Yellow 之间添加 Green。

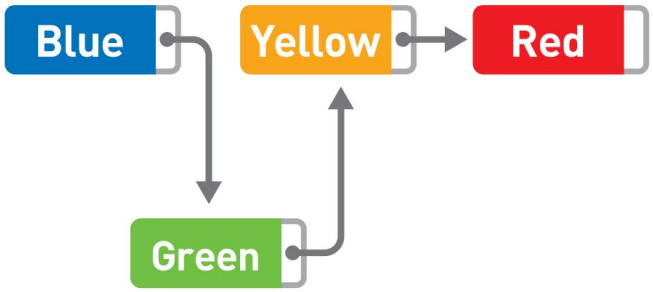


图 6.　　重新构建指针关系

将 Blue 的指针指向的位置变成 Green，然后再把 Green 的指针指向 Yellow，数据的添加就大功告成了。

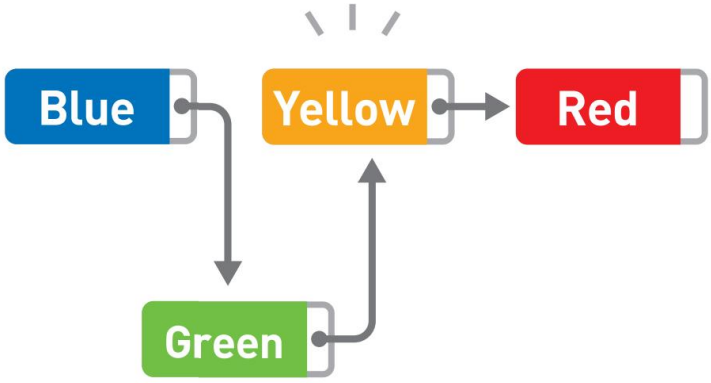


图 7.　　删除单链表元素

数据的删除也一样，只要改变指针的指向就可以，比如删除 Yellow。

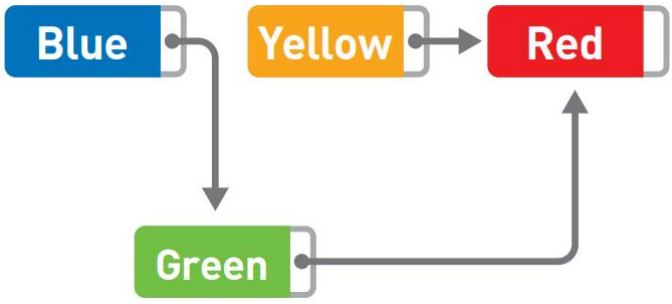


图 8.　　删除单链表元素重新构建指针关系

这时，只需要把 Green 指针指向的位置从 Yellow 变成 Red，删除就完成了。虽然 Yellow 本身还存储在内存中，但是不管从哪里都无法访问这个数据，所以也就没有特意去删除它的必要了。今后需要用到 Yellow 所在的存储空间时，只要用新数据覆盖掉就可以了。

对链表的操作所需的运行时间到底是多少呢？在这里，我们把链表中的数据量记成 n。访问数据时，我们需要从链表头部开始查找（线性查找），如果目标数据在链表最后的话，需要的时间就是 O(n)。

另外，添加数据只需要更改两个指针的指向，所以耗费的时间与 n 无关。如果已经到达了添加数据的位置，那么添加操作只需花费 O(1) 的时间。删除数据同样也只需 O(1) 的时间。

2. 循环链表

3. 双向链表

五. 数组

数组也是数据呈线性排列的一种数据结构。与前面的链表不同，在数组中，访问数据十分简单，而添加和删除数据比较耗工夫。

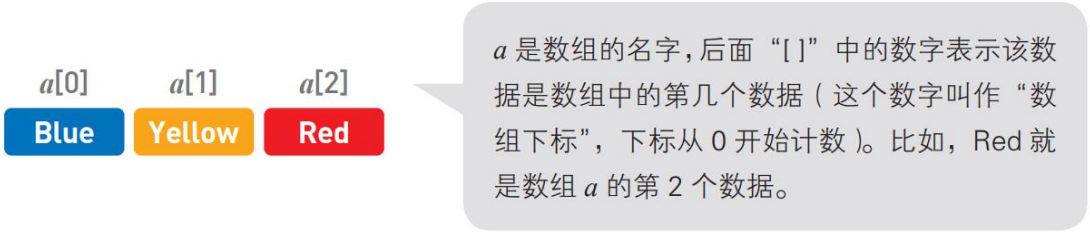


图 9.　　数组

这就是数组的概念图。Blue、Yellow、Red 作为数据存储在数组中。

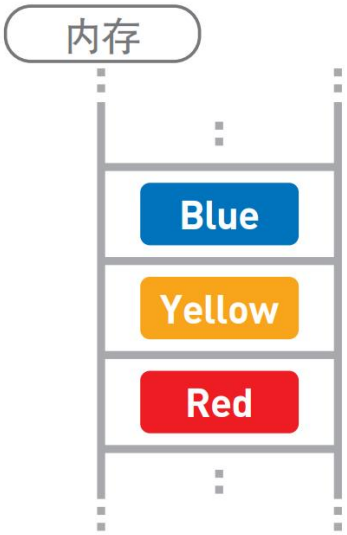


图 10.　数组中数据的存储空间是连续的

数据按顺序存储在内存的连续空间内。

图 11.　数组可直接访问目标数据

由于数据是存储在连续空间内的，所以每个数据的内存地址（在内存上的位置）都可以通过数组下标算出，我们也就可以借此直接访问目标数据（这叫作“随机访问”）

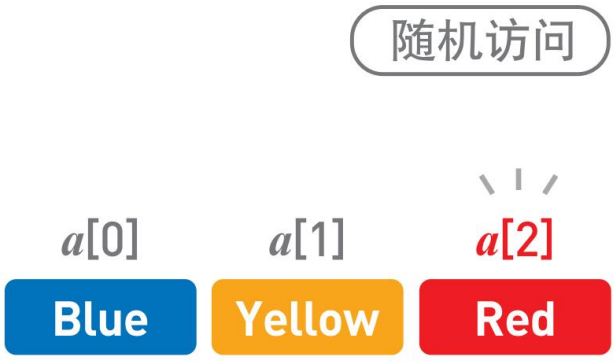


图 12.　访问数据时可随机访问

比如现在我们想要访问 Red。如果使用指针（链表）就只能从头开始查找，但在数组中，只需要指定 a[2]，便能直接访问 Red。

图 13.　准备在数组中添加数据

但是，如果想在任意位置上添加或者删除数据，数组的操作就要比链表复杂多了。这里我们尝试将 Green 添加到第 2 个位置上。

图 14.　新分配一个存储空间

首先，在数组的末尾确保需要增加的存储空间。

为了给新数据腾出位置，要把已有数据一个个移开。

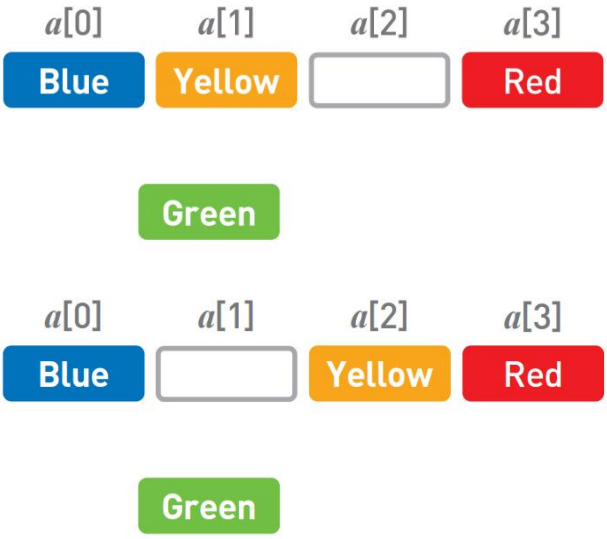


图 15.　将已有数据依次后移腾出插入的空间

首先把 Red 往后移，然后把 Yellow 往后移。

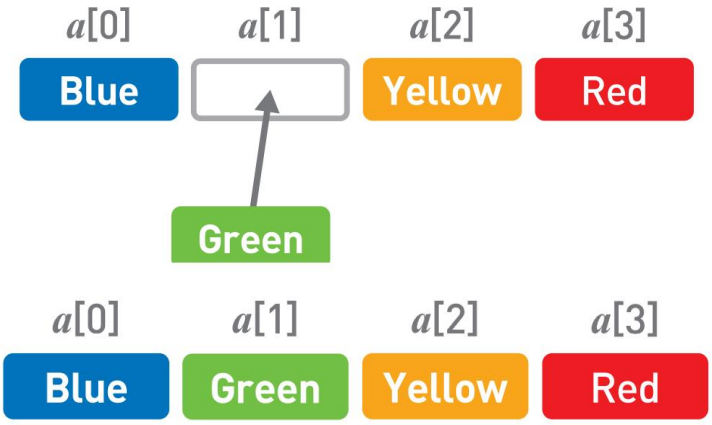


图 16.　将数据插入到空余的位置

最后在空出来的位置上写入 Green，完成添加数据的操作。

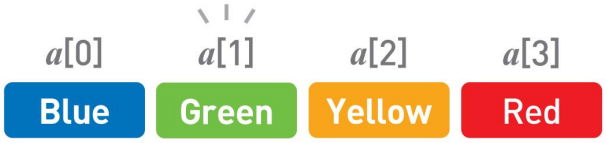


图 17.　删除数组元素

反过来，如果想要删除数组元素 Green……

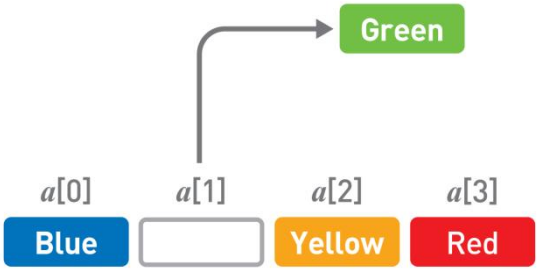


图 18.　将目标元素移出数组

首先，删掉目标数据（在这里指 Green）。

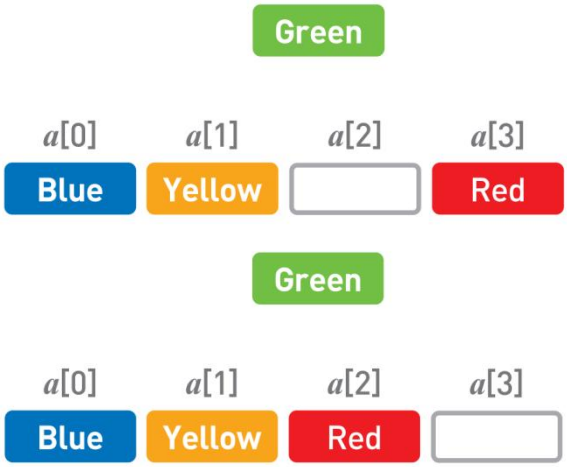
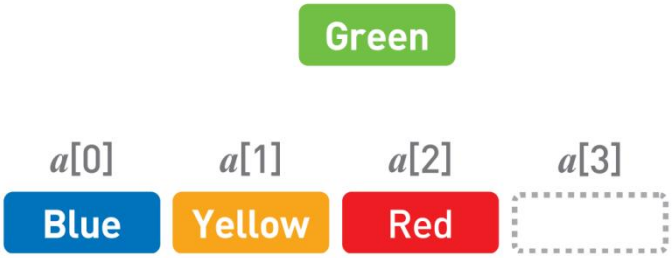


图 19.　剩余数据前移，补齐空位

然后把后面的数据一个个往前移，以补齐空位。



最后再删掉多余的空间。这样一来 Green 便被删掉了。

这里讲解一下对数组操作所花费的运行时间。假设数组中有 n 个数据，由于访问数据时使用的是随机访问（通过下标可计算出内存地址），所以需要的运行时间仅为恒定的 O(1)。

但另一方面，想要向数组中添加新数据时，必须把目标位置后面的数据一个个移开。所以，如果在数组头部添加数据，就需要 O(n) 的时间。删除操作同理。

1. 补充说明

在链表和数组中，数据都是线性地排成一列。在链表中访问数据较为复杂，添加和删除数据较为简单；而在数组中访问数据比较简单，添加和删除数据却比较复杂。

我们可以根据哪种操作较为频繁来决定使用哪种数据结构。

	访问	添加	删除
链表	慢	快	快
数组	快	慢	慢

六. 栈

栈也是一种数据呈线性排列的数据结构，不过在这种结构中，我们只能访问最新添加的数据。栈就像是一摞书，拿到新书时我们会把它放在书堆的最上面，取书时也只能从最上面的新书开始取。

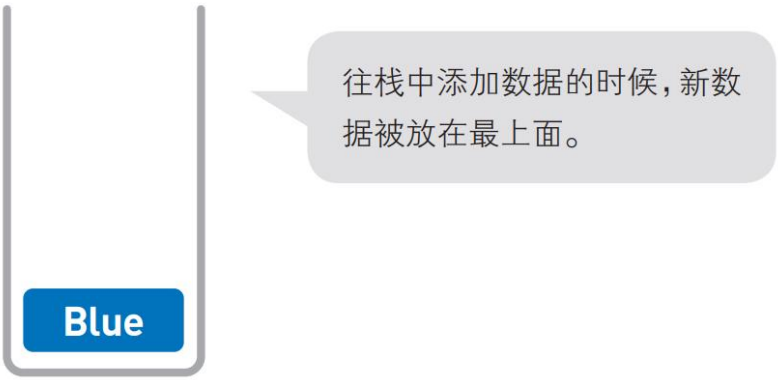


图 20.　栈的结构图

上图就是栈的概念图，可以看到，现在存储在栈中的只有数据 Blue。

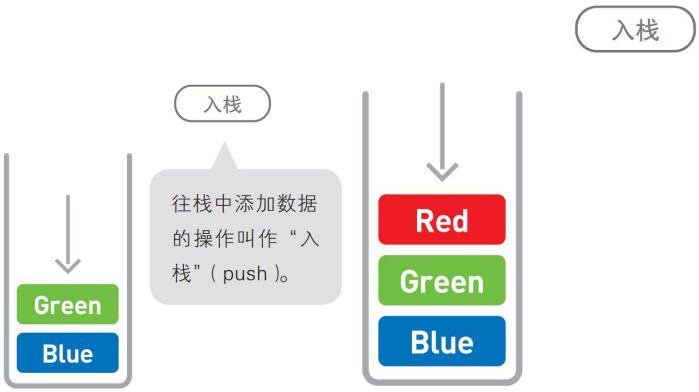


图 21.　入栈

如上图所示，当我们需要在栈中添加数据时，我们只能将其添加到最上面，而不能是其他位置。

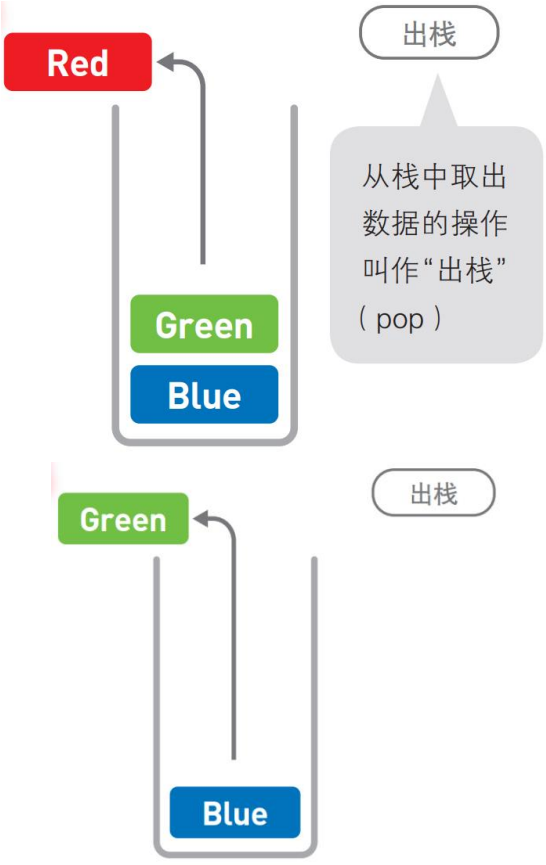


图 22.　出栈

从栈中取出数据时，是从最上面，也就是最新的数据开始取出的。这里取出的是 Red。如果再进行一次出栈操作，取出的就是 Green 了。

像栈这种最后添加的数据最先被取出，即“后进先出”的结构，我们称为 Last In First Out，简称 LIFO。

与链表和数组一样，栈的数据也是线性排列，但在栈中，添加和删除数据的操作只能在一端进行，访问数据也只能访问到顶端的数据。想要访问中间的数据时，就必须通过出栈操作将目标数据移到栈顶才行。

栈只能在一端操作这一点看起来似乎十分不便，但在只需要访问最新数据时，使用它就比较方便了。

比如，规定（AB（C（DE）F）（G（（H）I J）K））这一串字符中括号的处理方式

如下：首先从左边开始读取字符，读到左括号就将其入栈，读到右括号就将栈顶的左括号出栈。此时，出栈的左括号便与当前读取的右括号相匹配。通过这种处理方式，我们就能得知配对括号的具体位置。

七. 队列

与前面提到的数据结构相同，队列中的数据也呈线性排列。虽然与栈有些相似，但队列中添加和删除数据的操作分别是在两端进行的。就和“队列”这个名字一样，把它想象成排成一队的人更容易理解。在队列中，处理总是从第一名开始往后进行，而新来的人只能排在队尾。

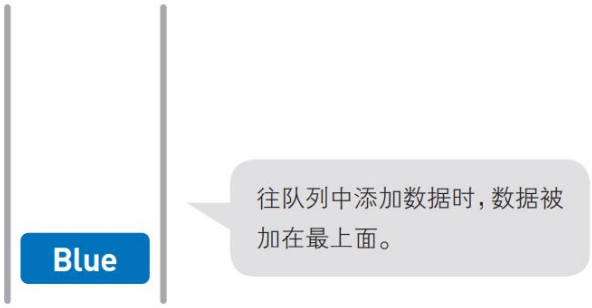
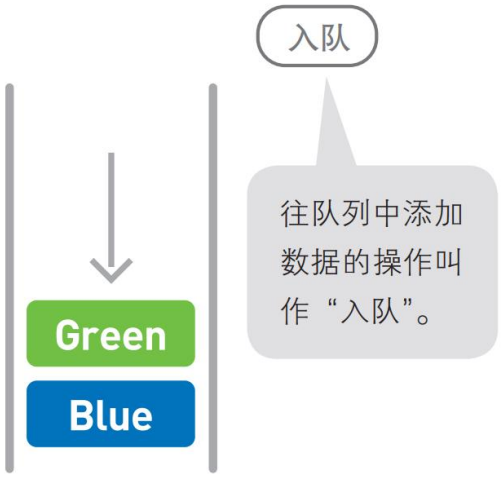


图 23. 队列示意图

上图就是队列的概念图，现在队列中只有数据 Blue。



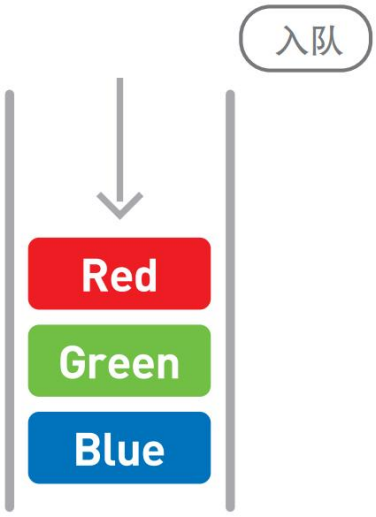


图 24.　入队

由上图可以得知，向队列中添加新的数据，这些数据只能添加到末尾，而不能是其他位置。

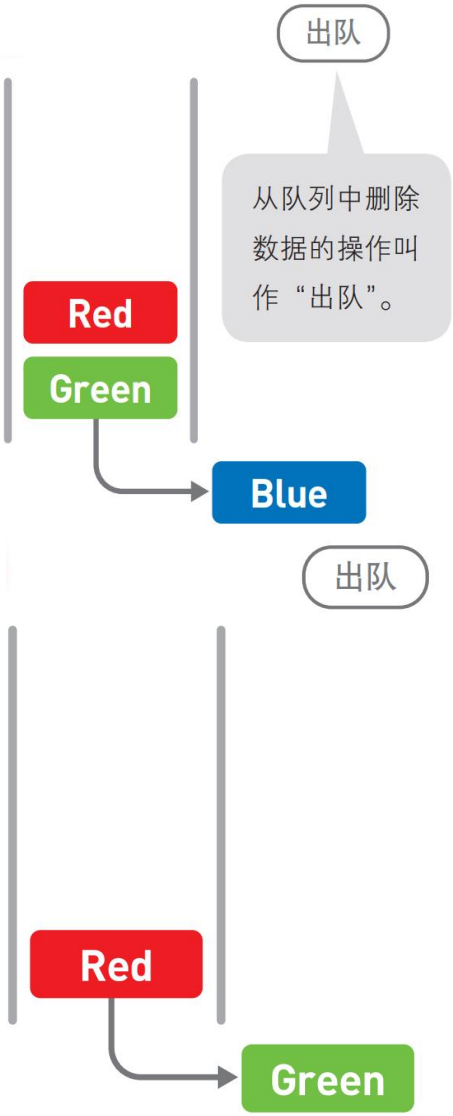


图 25.　出队列

从队列中取出（删除）数据时，是从最下面，也就是最早入队的数据开始的。第一张图取出的是 Blue，如果再进行一次出队操作，取出的就是 Green 了。

像队列这种最先进去的数据最先被取来，即“先进先出”的结构，我们称为 First In First Out，简称 FIFO。

与栈类似，队列中可以操作数据的位置也有一定的限制。在栈中，数据的添加和删除都在同一端进行，而在队列中则分别是在两端进行的。队列也不能直接访问位于中间的数据，必须通过出队操作将目标数据变成首位后才能访问。

八. 哈希表

在哈希表这种数据结构中，可以使数据的查询效率得到显著提升。

图 26. 哈希表

哈希表存储的是由键（key）和值（value）组成的数据。例如，我们将每个人的性别作为数据进行存储，键为人名，值为对应的性别。

一般来说，我们可以把键当成数据的标识符，把值当成数据的内容。

图 27. 存储在数组中

为了和哈希表进行对比，我们先将这些数据存储在数组中。

图 28.　每一个数据对应的下标

此处准备了 6 个箱子（即长度为 6 的数组）来存储数据。假设我们需要查询 Ally 的性别，由于不知道 Ally 的数据存储在哪个箱子里，所以只能从头开始查询。这个操作便叫作“线性查找”。

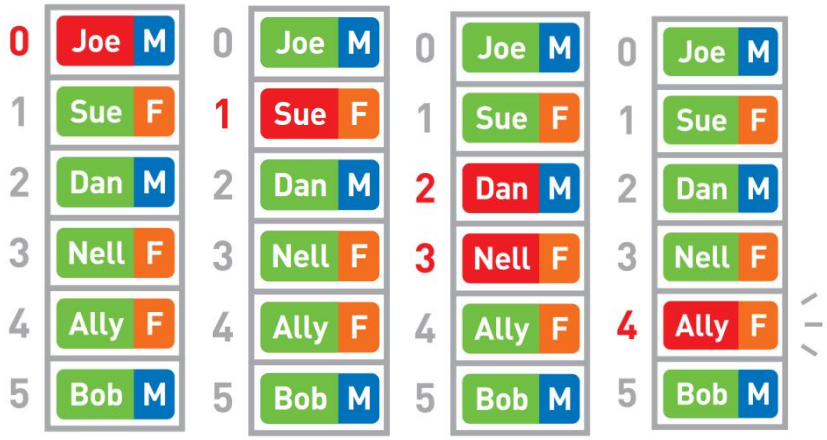


图 29.　线性查找

0 号箱子中存储的键是 Joe 而不是 Ally，1 号箱子中的也不是 Ally。同样，2 号、3 号箱子中的也都不是 Ally，查找到 4 号箱子的时候，发现其中数据的键为 Ally。把键对应的值取出，我们就知道 Ally 的性别为女（F）了。

数据量越多，线性查找耗费的时间就越长。由此可知 ：由于数据的查询较为耗时，所以此处并不适合使用数组来存储数据。

但使用哈希表便可以解决这个问题。

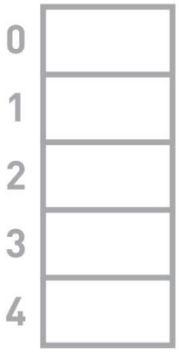


图 30.　准备存储数据用的数组

首先准备好数组，这次我们用 5 个箱子的数组来存储数据。

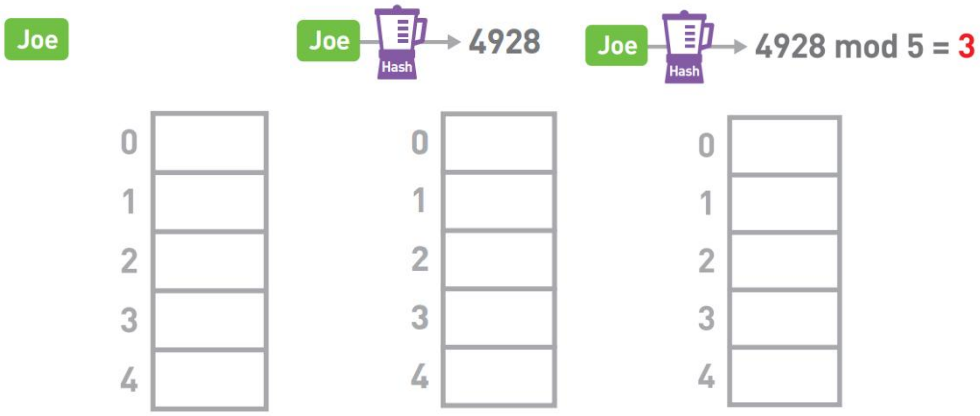


图 31.　计算哈希值及数据存储位置

尝试把 Joe 这个字符串存进去。使用哈希函数（Hash）计算 Joe 的键，也就是字符串“Joe”的哈希值，得到的结果为 4928。

将得到的哈希值除以数组的长度 5，求得其余数。这样的求余运算叫作“mod 运算”。此处 mod 运算的结果为 3。

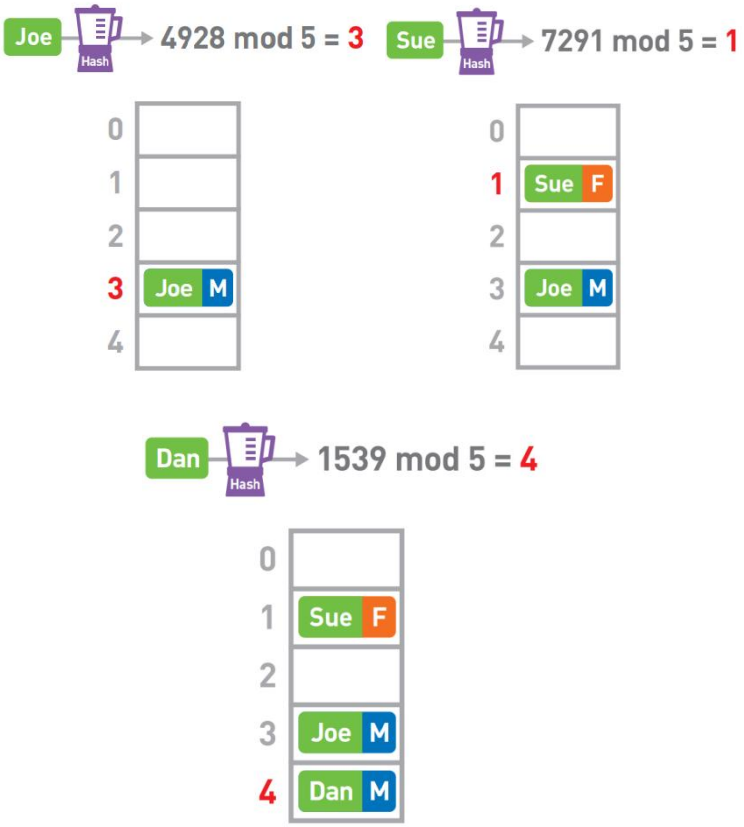


图 32.　将对应的字符串存储到特定位置

因此，我们将字符串 Joe 的数据存进数组的 3 号箱子中。重复前面的操作，将其他数据也存进数组中。Sue 键的哈希值为 7291，mod 5 的结果为 1，将 Sue 的数据存进 1 号箱中。Dan 键的哈希值为 1539，mod 5 的结果为 4，将 Dan 的数据存进 4 号箱中。

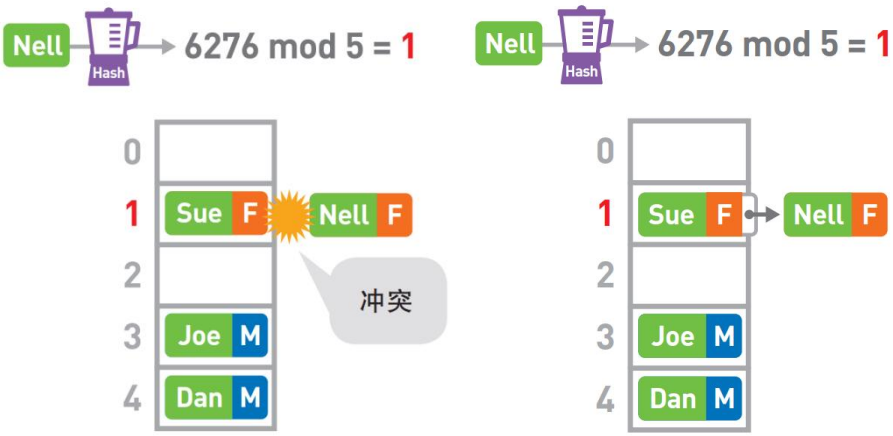


图 33.　存储位置重复发生冲突

Nell 键的哈希值为 6276，mod 5 的结果为 1。本应将其存进数组的 1 号箱中，但此时 1 号箱中已经存储了 Sue 的数据。这种存储位置重复了的情况便叫作“冲突”。遇到这种情况，可使用链表在已有数据的后面继续存储新的数据。

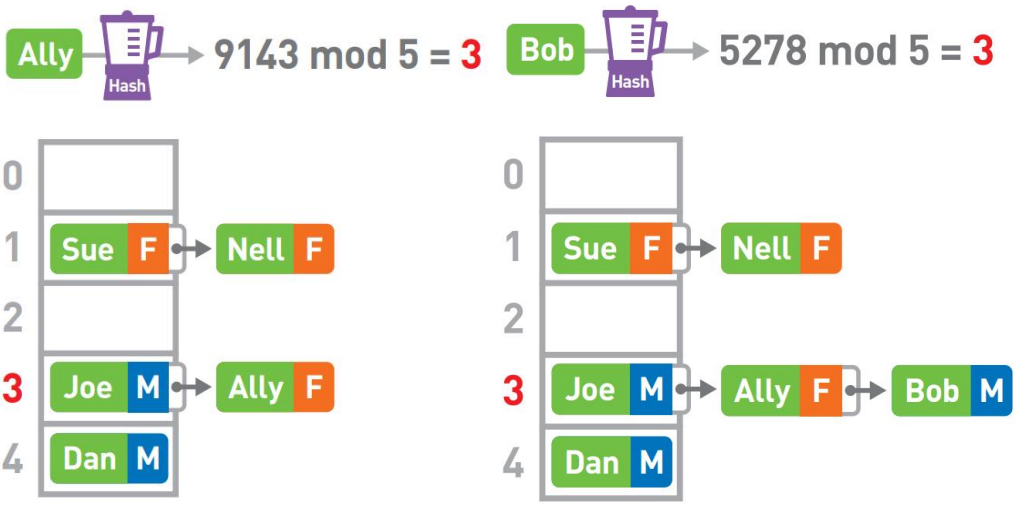


图 34.　解决哈希冲突

Ally 键的哈希值为 9143，mod 5 的结果为 3。本应将其存储在数组的 3 号箱中，但 3 号箱中已经有了 Joe 的数据，这里又发生了“哈希冲突”，所以使用链表，在其后面存储 Ally 的数据。Bob 键的哈希值为 5278，mod 5 的结果为 3。本应将其存储在数组的 3 号箱中，但 3 号箱中已经有了 Joe 和 Ally 的数据，这里又发生了“哈希冲突”，所以使用链表，在 Ally 的后面继续存储 Bob 的数据。

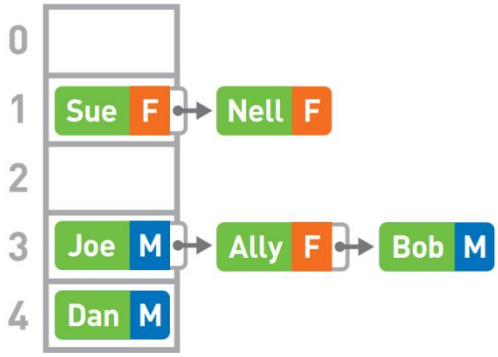
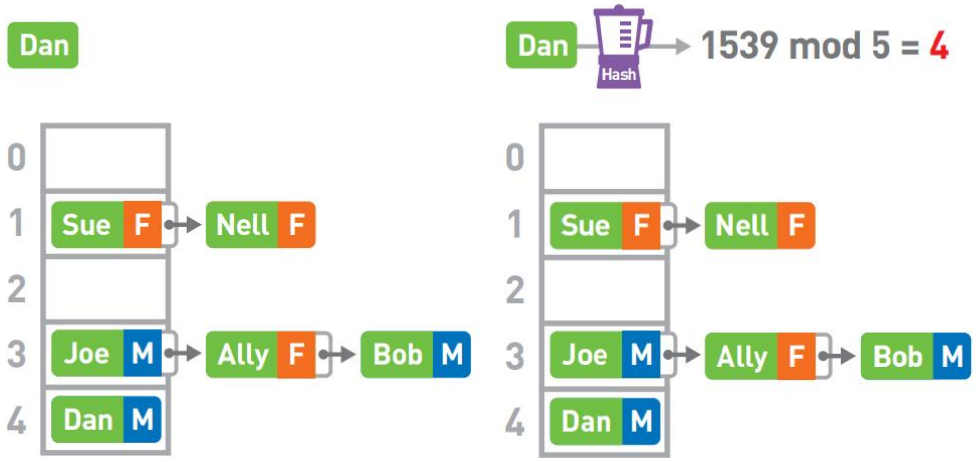


图 35.　完成哈希表

像这样存储完所有数据，哈希表也就制作完成了。



接下来讲解数据的查询方法。假设我们要查询 Dan 的性别，为了知道 Dan 存储在哪个箱子里，首先需要算出 Dan 键的哈希值，然后对其进行 mod 运算。最后得到的结果为 4，于是我们知道了它存储在 4 号箱中。

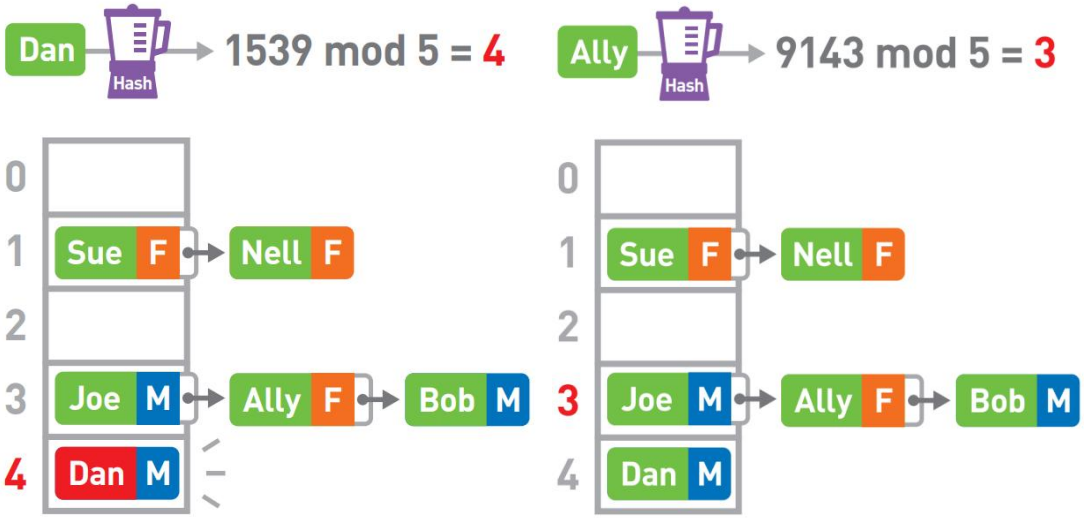


图 36.　根据键值进行查找

查看 4 号箱可知，其中的数据的键与 Dan 一致，于是取出对应的值。由此我们便知道了 Dan 的性别为男（M）。那么，想要查询 Ally 的性别时该怎么做呢？为了找到它的存储位置，先要算出 Ally 键的哈希值，再对其进行 mod 运算。最终得到的结果为 3。

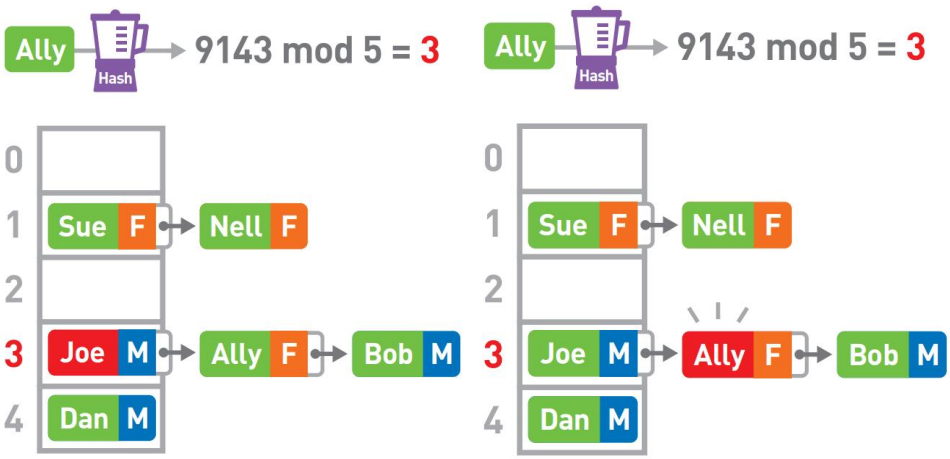


图 37.　当对应键值的数据不对时，需要进行线性查找

然而 3 号箱中数据的键是 Joe 而不是 Ally，此时便需要对 Joe 所在的链表进行线性查找。于是我们找到了键为 Ally 的数据，取出其对应的值，便知道了 Ally 的性别为女（F）。

在哈希表中，我们可以利用哈希函数快速访问到数组中的目标数据。如果发生哈希冲突，就使用链表进行存储。这样一来，不管数据量为多少，我们都能够灵活应对。

如果数组的空间太小，使用哈希表的时候就容易发生冲突，线性查找的使用频率也会更高；反过来，如果数组的空间太大，就会出现很多空箱子，造成内存的浪费。因此，给数组设定合适的空间非常重要。

1. 解决哈希冲突

在存储数据的过程中，如果发生冲突，可以利用链表在已有数据的后面插入新数据来解决冲突。这种方法被称为“链地址法”。

除了链地址法以外，还有几种解决冲突的方法。其中，应用较为广泛的是“开放地址法”。这种方法是指当冲突发生时，立刻计算出一个候补地址（数组上的位置）并将数据存进去。如果仍然有冲突，便继续计算下一个候补地址，直到有空地址为止。可以通过多次使用哈希函数或“线性探测法”等方法计算候补地址。

1) 开放地址法

从发生冲突的那个单元起，按照一定的次序，从哈希表中找到一个空闲的单元。然后把发生冲突的元素存入到该单元的一种方法。开放地址法需要的表长度要大于等于所需要存放的元素。

在开放地址法中解决冲突的方法有：线性探查法、平方探查法、双散列函数探查法。

开放地址法的缺点在于删除元素的时候不能真的删除，否则会引起查找错误；只能做一个特殊标记，直到有下个元素插入才能真正删除该元素。

a. 线性探查法

线性探查法是开放定址法中最简单的冲突处理方法，它从发生冲突的单元起，依次判断下一个单元是否为空，当达到最后一个单元时，再从表首依次判断。直到碰到空闲的单元或者探查完全部单元为止。

b. 平方探查法

平方探查法即是发生冲突时，用发生冲突的单元 d[i], 加上 1^2、 2^2 等。即 $d[i] + 1^2$，$d[i] + 2^2$, $d[i] + 3^2$ …直到找到空闲单元。

在实际操作中，平方探查法不能探查到全部剩余的单元。不过在实际应用中，能探查到一半单元也就可以了。若探查到一半单元仍找不到一个空闲单元，表明此散列表太满，应该重新建立。

c. 双散列函数探查法

这种方法使用两个散列函数 h1 和 h2。其中 h1 和前面的 h 一样，以关键字为自变量，产生一个 0 至 m—l 之间的数作为散列地址；h2 也以关键字为自变量，产生一个 l 至 m—1 之间的、并和 m 互素的数(即 m 不能被该数整除)作为探查序列的地址增量(即步长)，探查序列的步长值是固定值 l；对于平方探查法，探查序列的步长值是探查次数 i 的两倍减 l；对于

双散列函数探查法，其探查序列的步长值是同一关键字的另一散列函数的值。

2) 链地址法

链接地址法的思路是将哈希值相同的元素构成一个同义词的单链表，并将单链表的头指针存放在哈希表的第 i 个单元中，查找、插入和删除主要在同义词链表中进行。链表法适用于经常进行插入和删除的情况。

3) 再哈希法

就是同时构造多个不同的哈希函数：

Hi = RHi(key)　　i= 1,2,3 … k;

当 H1 = RH1(key) 发生冲突时，再用 H2 = RH2(key) 进行计算，直到冲突不再产生，这种方法不易产生聚集，但是增加了计算时间。

九. 堆

堆是一种图的树形结构，被用于实现“优先队列”（priority queues）。优先队列是一种数据结构，可以自由添加数据，但取出数据时要从最小值开始按顺序取出。在堆的树形结构中，各个顶点被称为“结点”（node），数据就存储在这些结点中。

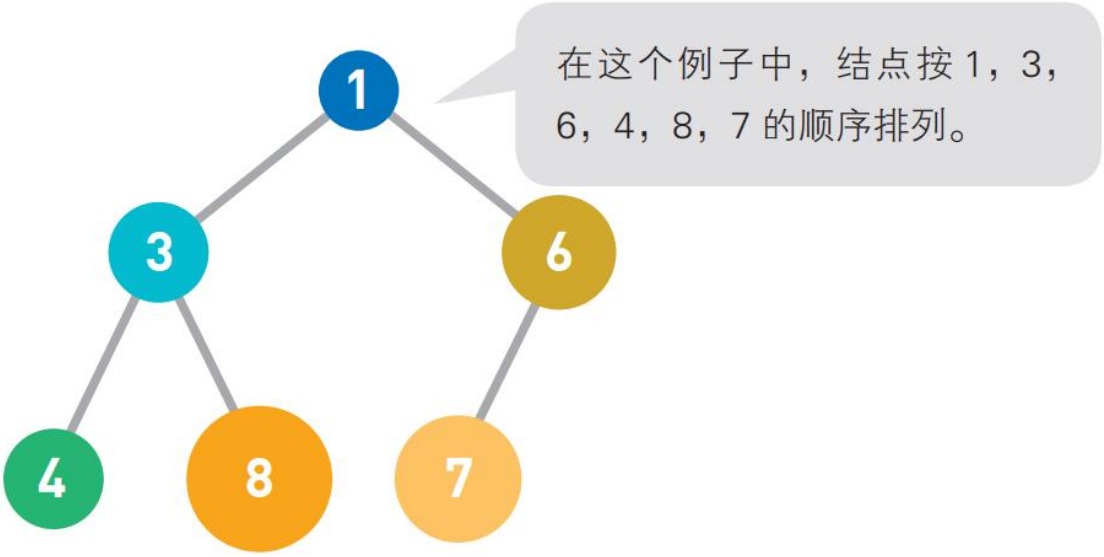


图 38.　堆的示例

结点内的数字就是存储的数据。堆中的每个结点最多有两个子结点，树的形状取决于数据的个数。另外，结点的排列顺序为从上到下，同一行里则为从左到右。

在堆中存储数据时必须遵守这样一条规则 ：子结点必定大于父结点。因此，最小值被存储在顶端的根结点中。

往堆中添加数据时，为了遵守这条规则，一般会把新数据放在最下面一行靠左的位置。当最下面一行里没有多余空间时，就再往下另起一行，把数据加在这一行的最左端。

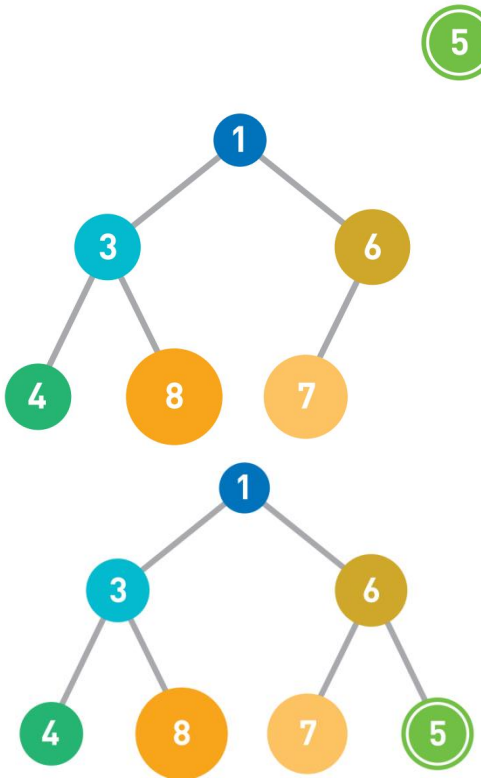


图 39.　向堆中添加数据

我们试试往堆里添加数字 5。首先按照上面的规则寻找新数据的位置，该图中最下面一排空着一个位置，所以将数据加在此处。

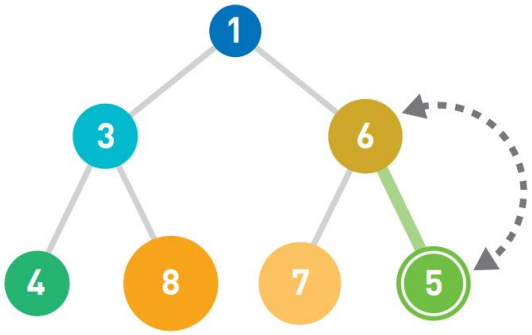


图 40.　判断插入位置是否正确

如果父结点大于子结点，则不符合上文提到的规则，因此需要交换父子结点的位置。这里由于父结点的 6 大于子结点的 5，所以交换了这两个数字。重复这样的操作直到数据都符合规则，不再需要交换为止。

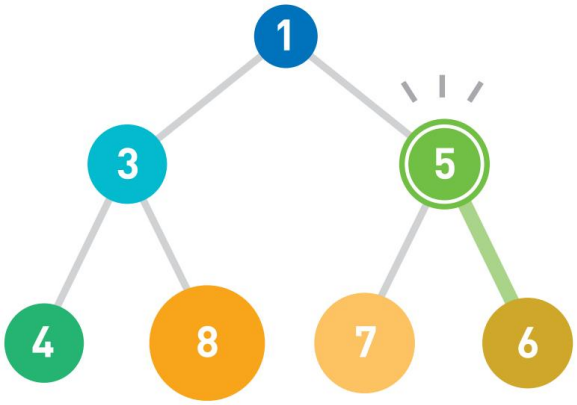


图 41.　判断插入位置是否正确

现在，父结点的 1 小于子结点的 5，父结点的数字更小，所以不再交换。这样，往堆中添加数据的操作就完成了。

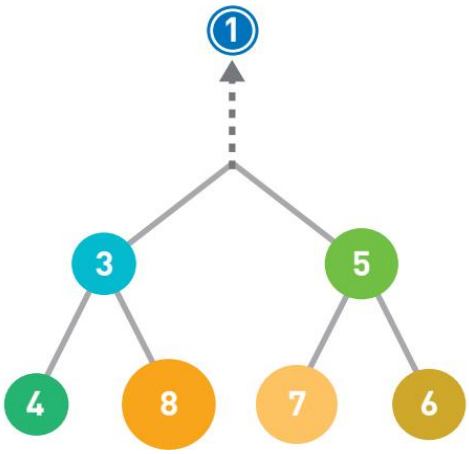


图 42.　从堆中取出数据

从堆中取出数据时，取出的是最上面的数据。这样，堆中就能始终保持最上面的数据最小。

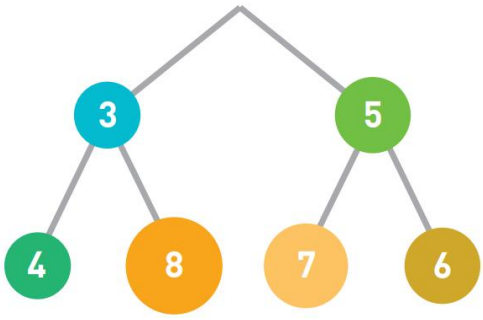


图 43.　取出数据后需要准备调整结构

由于最上面的数据被取出，因此堆的结构也需要重新调整。

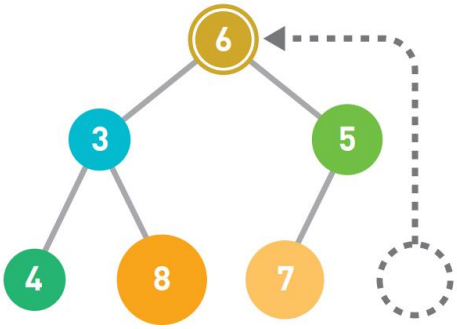


图 44.　移动数据

按照前面说明的排列顺序，将最后的数据（此处为 6）移动到最顶端。

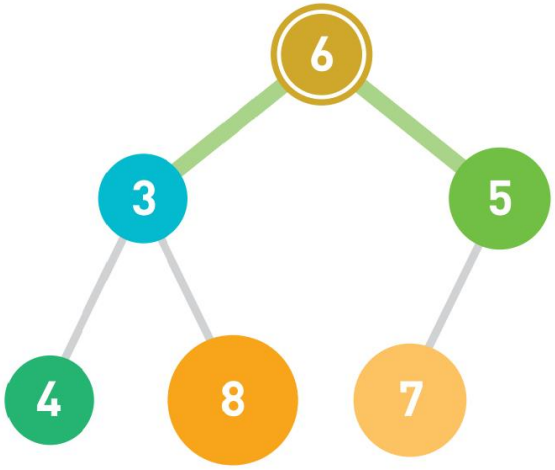


图 45.　判断调整后的结构是否正确

如果子结点的数字小于父结点的，就将父结点与其左右两个子结点中较小的一个进行交换。

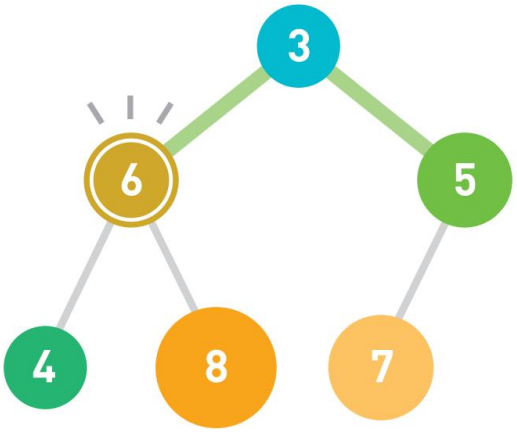


图 46.　不符合规则就按照规则移动

这里由于父结点的 6 大于子结点（右）的 5 大于子结点（左）的 3，所以将左边的子结点与父结点进行交换。重复这个操作直到数据都符合规则，不再需要交换为止。

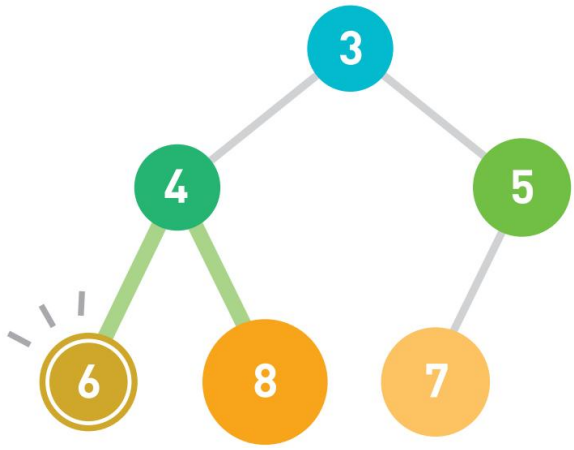


图 47.　完成堆的重构

现在，子结点（右）的 8 大于父结点的 6 大于子结点（左）的 4，需要将左边的子结点与父结点进行交换。这样，从堆中取出数据的操作便完成了。

1. 总结

堆中最顶端的数据始终最小，所以无论数据量有多少，取出最小值的时间复杂度都为 O(1)。

另外，因为取出数据后需要将最后的数据移到最顶端，然后一边比较它与子结点数据的大小，一边往下移动，所以取出数据需要的运行时间和树的高度成正比。假设数据量为 n，根据堆的形状特点可知树的高度为 log2n ，那么重构树的时间复杂度便为 O(logn)。

添加数据也一样。在堆的最后添加数据后，数据会一边比较它与父结点数据的大小，一边往上移动，直到满足堆的条件为止，所以添加数据需要的运行时间与树的高度成正比，也是 O(logn)。

如果需要频繁地从管理的数据中取出最小值，那么使用堆来操作会非常方便。比如狄克斯特拉算法，每一步都需要从候补顶点中选择距离起点最近的那个顶点。此时，在顶点的选择上就可以用到堆。

十. 树

1. 树的基本概念

1) 树的定义

任意一颗非空树应满足：有且仅有一个特定的称为根的结点。

树的定义是递归的，是一种递归的数据结构。树作为一种逻辑结构，同时也是一种分层结构，具有以下两个特点：

① 树的根结点没有前驱结点，除根结点外的所有结点有且仅有一个前驱结点。

② 树中所有节点可以有零个或多个后继结点。

n 个结点的树中有 n-1 条边。

2) 树的基本术语

① 根是树中唯一没有双亲的结点。

② 树中结点最大的度数称为树的度。

③ 结点的深度是从根结点开始自顶向下逐层累加的；树的高度是从叶结点开始自底向上逐层累加的；树的高度是树中结点的最大层数。

④ 树中结点的子树从左到右是有次序的，不能交换，这样的树称为有序树，有序树中，一个结点的子结点按从左到右的顺序出现是有关联的，反之则称为无序树。

⑤ 树中两个结点之间的路径是由这两个结点之间所经过的结点序列构成的，而路径长度是路径上所有经过的边的个数。

3) 树的性质

① 树中的结点数等于所有结点的度数加 1。

② 度为 m 的树中第 i 层上至多有m^{i-1}个结点（$i \geq 1$）。

③ 高度为 h 的 m 叉树至多有$\frac{(m^h-1)}{(m-1)}$个结点。

④ 具有 n 个结点的 m 叉树的最小高度为$\log_m (n(m-1)+1)$。

2. 二叉树

1) 二叉树的定义

二叉查找树（又叫作二叉搜索树或二叉排序树）是一种数据结构，采用了图的树形结构，数据存储于二叉查找树的各个结点中。二叉树的子树有左右之分，其次序不能任意颠倒，是有序树。即使树中结点只有一颗子树，也要区分它是左子树还是右子树。

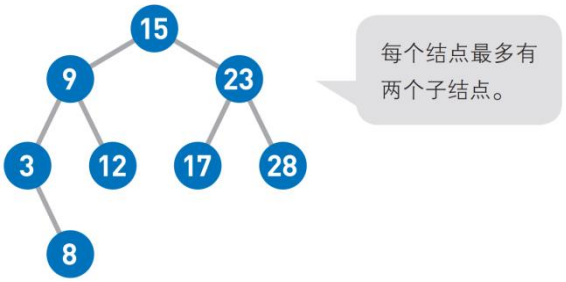


图 48. 二叉查找树的示例

结点中的数字便是存储的数据。此处以不存在相同数字为前提进行说明。

2) 二叉树与度为 2 的有序数的区别

① 度为 2 的树至少有 3 个结点，而二叉树可以为空。

② 度为 2 的有序树的孩子结点的左右次序是相对另一个孩子结点而言的，而二叉树无论其孩子数是否为 2，都要区分左右次序。

3) 二叉树的性质

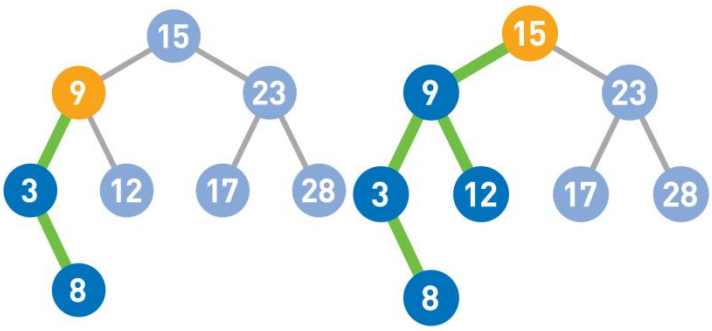


图 49. 二叉查找树性质

二叉查找树有两个性质。

第一个是每个结点的值均大于其左子树上任意一个结点的值。比如结点 9 大于其左子树上的 3 和 8。同样，结点 15 也大于其左子树上任意一个结点的值。

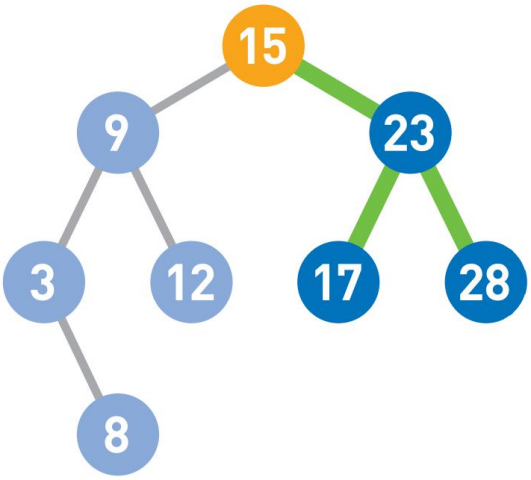


图 50.　二叉查找树性质

第二个是每个结点的值均小于其右子树上任意一个结点的值。比如结点 15 小于其右子树上的 23、17 和 28。

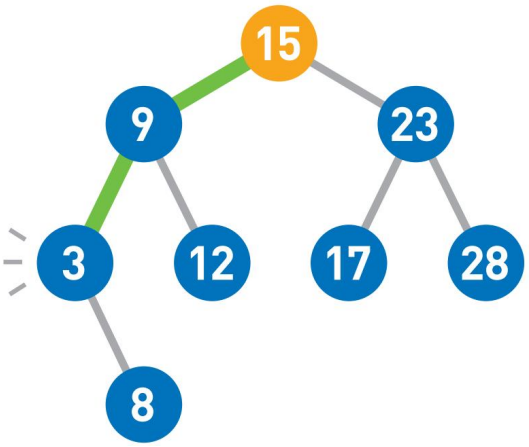


图 51.　查找最小节点

根据这两个性质可以得到以下结论。首先，二叉查找树的最小结点要从顶端开始，往其左下的末端寻找。此处最小值为 3。

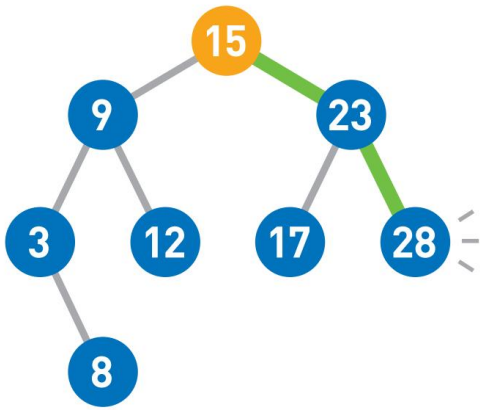


图 52.　查找最大节点

反过来，二叉查找树的最大结点要从顶端开始，往其右下的末端寻找。此处最大值为 28。

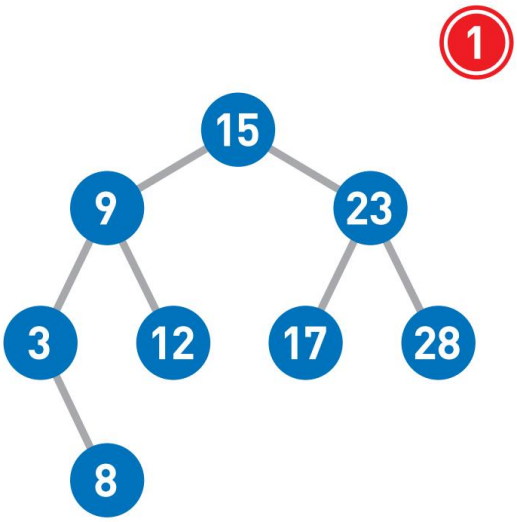


图 53.　向二叉查找树中添加数据

下面我们来试着往二叉查找树中添加数据。比如添加数字 1。

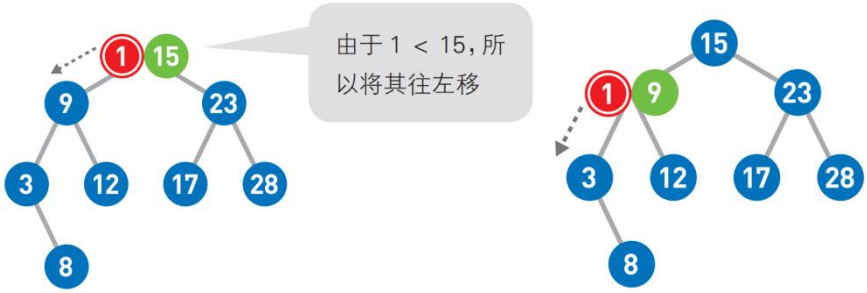


图 54.　寻找添加数字的位置

首先，从二叉查找树的顶端结点开始寻找添加数字的位置。将想要添加的 1 与该结点中的值进行比较，小于它则往左移，大于它则往右移。由于 1＜9，所以将 1 往左移。

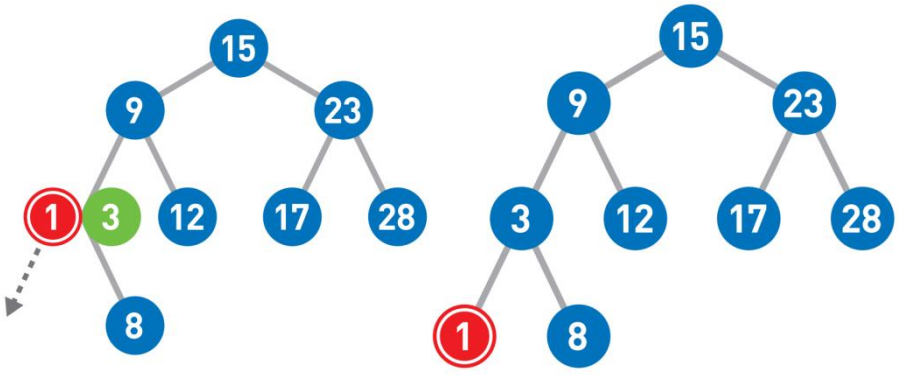


图 55.　完成添加操作

又由于 1＜3，所以继续将 1 往左移，但前面已经没有结点了，所以把 1 作为新结点添加到左下方。这样，1 的添加操作便完成了。

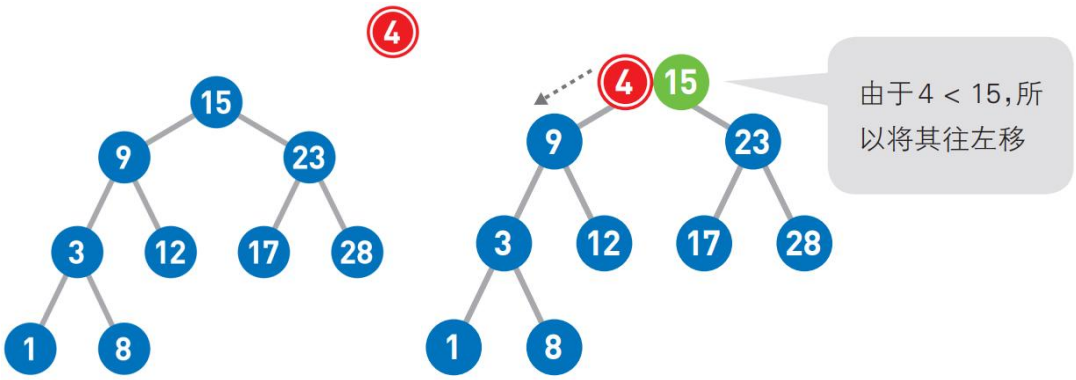
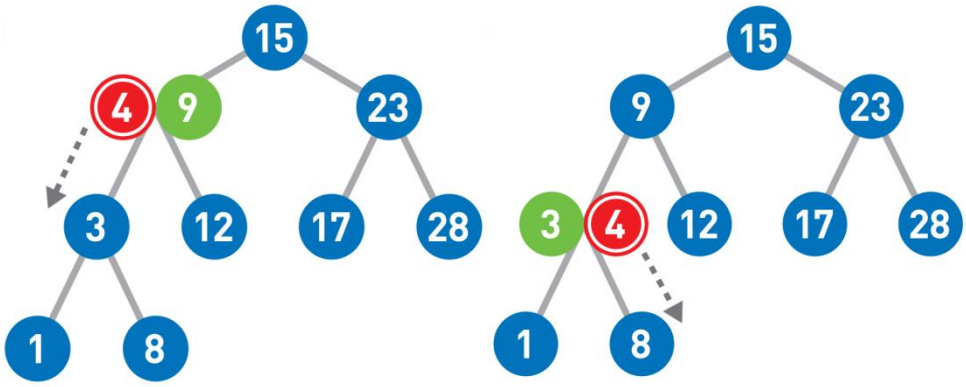


图 56.　　继续添加数据 4

接下来，我们再试试添加数字 4。和前面的步骤一样，首先从二叉查找树的顶端结点开始寻找添加数字的位置。



由于 4＜9，所以将其往左移。由于 4＞3，所以将其往右移。

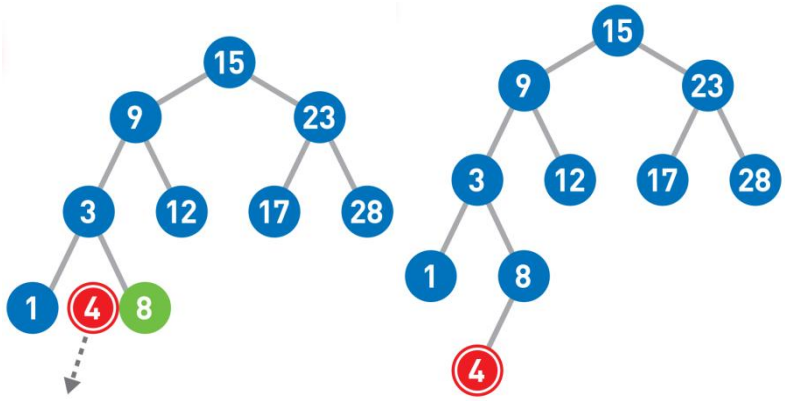


图 57.　　完成添加 4 的操作

由于 4＜8，所以需要将其往左移，但前面已经没有结点了，所以把 4 作为新结点添加到左下方。于是 4 的添加操作也完成了。

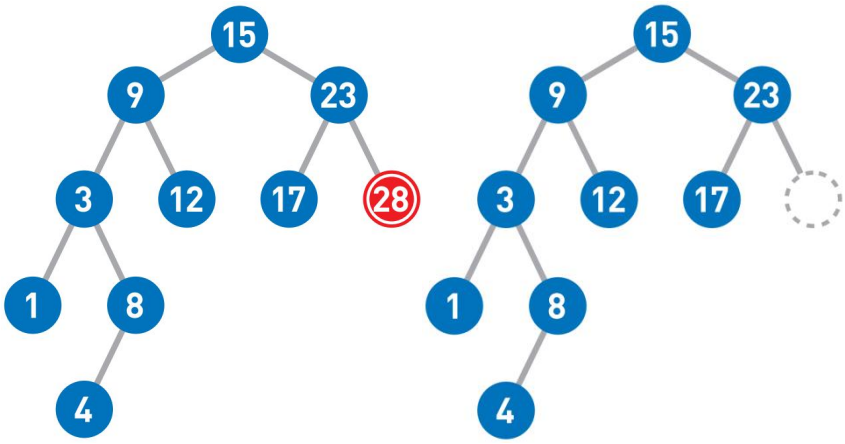


图 58.　删除的节点没有子节点，直接删除

接下来看看如何在二叉查找树中删除结点，比如我们来试试删除结点 28。如果需要删除的结点没有子结点，直接删掉该结点即可。

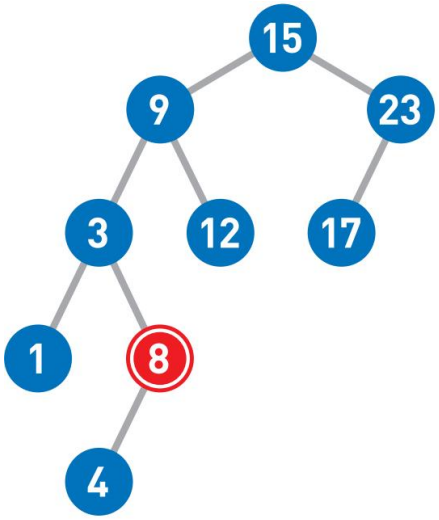


图 59.　准备删除节点 8

再试试删除结点 8。

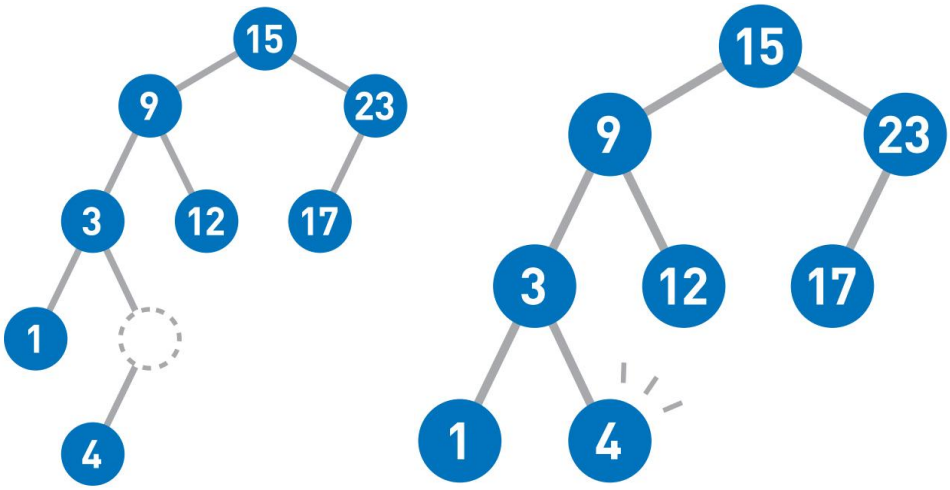


图 60.　节点只有一个子节点时进行删除

如果需要删除的结点只有一个子结点，那么先删掉目标结点，然后把子结点移到被删除结点的位置上即可。

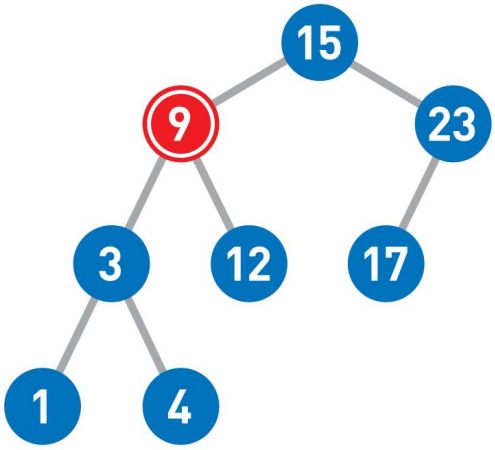


图 61.　删除具有两个子节点的节点

最后来试试删除结点 9，这个节点具有两个子节点。

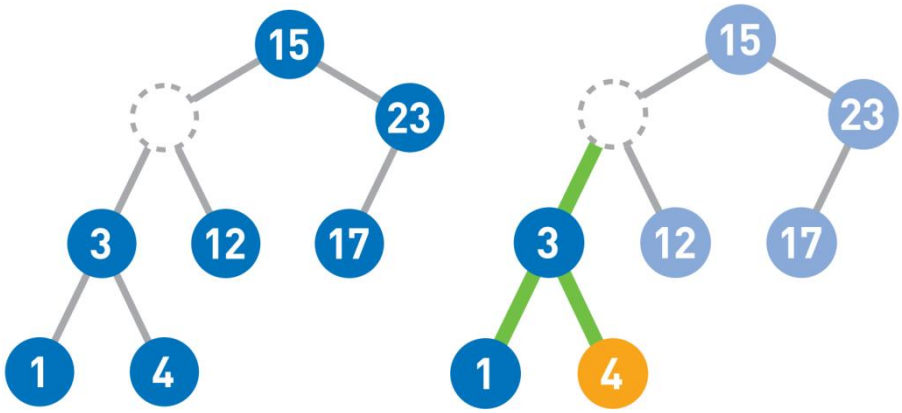


图 62.　删掉目标结点并寻找后继的最大结点

如果需要删除的结点有两个子结点，那么先删掉目标结点，然后在被删除结点的左子树中寻找最大结点。

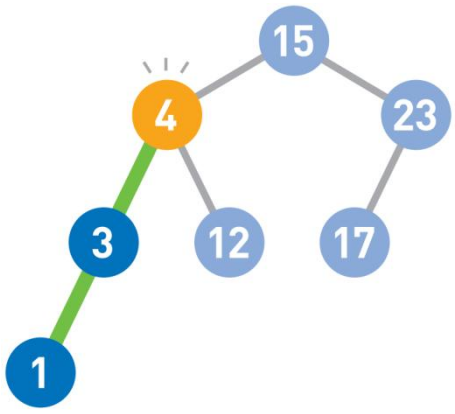


图 63.　将后继的最大结点移到被删除结点的位置上

最后将最大结点移到被删除结点的位置上。这样一来，就能在满足二叉查找树性质的前提下删除结点了。

如果需要移动的结点（此处为 4）还有子结点，就递归执行前面的操作。

删除 9 的时候，我们将“左子树中的最大结点”移动到了删除结点的位置上，但是根据二叉查找树的性质可知，移动“右子树中的最小结点”也没有问题。

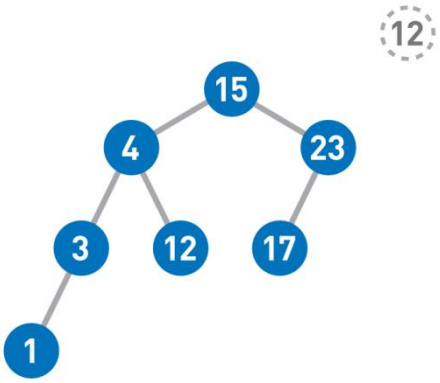


图 64.　查找节点

下面来看看如何在二叉查找树中查找结点，比如我们来试试查找 12。

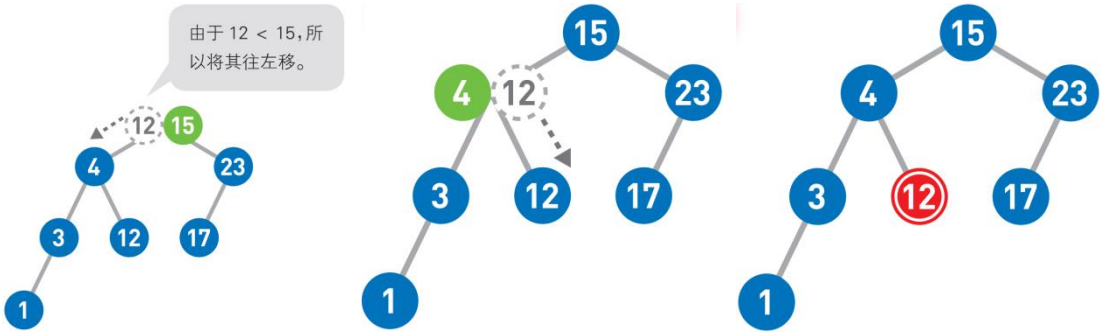


图 65.　从顶端结点开始往下查找

从二叉查找树的顶端结点开始往下查找。和添加数据时一样，把 12 和结点中的值进行比较，小于该结点的值则往左移，大于则往右移。由于 12＞4，所以往右移，便找到结点 12 了。

4)　总结

我们可以把二叉查找树当作是二分查找算法思想的树形结构体现。因为它具有前面提到的那两个性质，所以在查找数据或寻找适合添加数据的位置时，只要将其和现有的数据比较大小，就可以根据比较结果得知该往哪边移动了。

比较的次数取决于树的高度。所以如果结点数为 n，而且树的形状又较为均衡的话，比较大小和移动的次数最多就是 log2n。因此，时间复杂度为 O(logn)。但是，如果树的形状朝单侧纵向延伸，树就会变得很高，此时时间复杂度也就变成了 O(n)。

有很多以二叉查找树为基础扩展的数据结构，比如“平衡二叉查找树”。这种数据结构可以修正形状不均衡的树，让其始终保持均衡形态，以提高查找效率。

另外，虽然文中介绍的二叉查找树中一个结点最多有两个子结点，但我们可以把子结点数扩展为 m（m 为预先设定好的常数）。像这种子结点数可以自由设定，并且形状均衡的树便是 B 树。

3. 四叉树

4. 八叉树

十一.　图

1. 离散数学中的图

说到“图”，可能大部分人想到的是饼状图、柱状图，或者数学中 y = f(x)所呈现的图，而计算机科学或离散数学中说的“图”却是下面这样的。

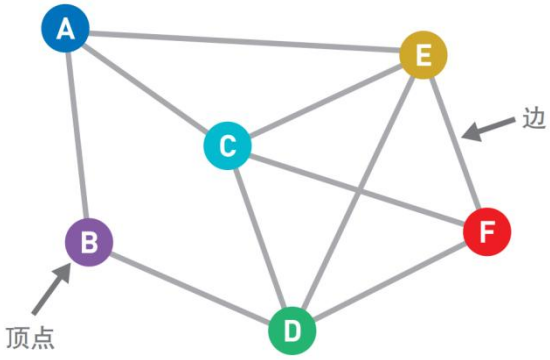


图 66.　计算机中的图结构

上图中的圆圈叫作“顶点”（也叫“结点”），连接顶点的线叫作“边”。也就是说，由顶点和连接每对顶点的边所构成的图形就是图。

2. 图可以表现各种关系

图可以表现社会中的各种关系，使用起来非常方便。假设我们要开一个派对，将参加人员作为顶点，把互相认识的人用边连接，就能用图来表现参加人员之间的人际关系了。

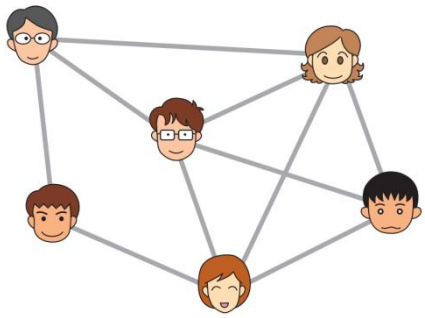


图 67.　用图表示关系

再举个例子，若将车站作为顶点，将相邻两站用边连接，就能用图来表现地铁的路线了。

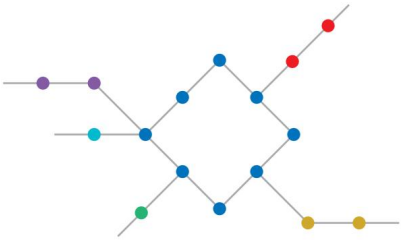


图 68.　用图表示地铁路线

另外，还可以在计算机网络中把路由器作为顶点，将相互连接的两个路由器用边连，这样就能用图来表现网络的连接关系了。

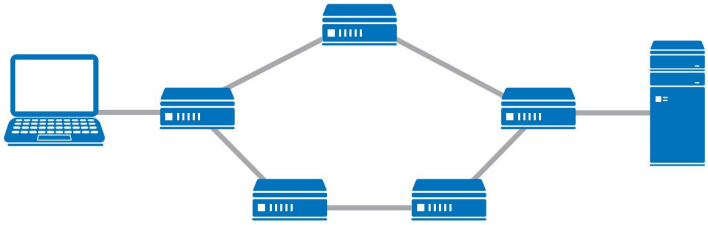


图 69.　表示网络的链接关系

3. 加权图

上面讲到的都是由顶点和边构成的图，而我们还可以给边加上一个值。

这个值叫作边的“权重”或者“权”，加了权的图被称为“加权图”。没有权的边只能表

示两个顶点的连接状态，而有权的边就可以额外表示顶点之间的“连接程度”。

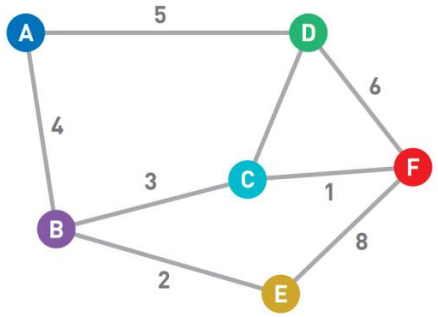


图 70.　　加权图

这个“程度”是什么意思呢？根据图的内容不同，“程度”表示的意思也不同。比如在计算机网络中，给两台路由器之间的边加上传输数据所需要的时间，这张图就能表示网络之间的通信时间了。

而在路线图中，如果把地铁在两个车站间行驶的时间加在边上，这张图就能表现整个路线的移动时间；如果把两个车站间的票价加在边上，就能表现乘车费了。

4. 有向图

当我们想在路线图中表示该路线只能单向行驶时，就可以给边加上箭头，而这样的图就叫作“有向图”。比如网页里的链接也是有方向性的，用有向图来表示就会很方便。

与此相对，边上没有箭头的图便是“无向图”。

下图中我们可以从顶点 A 到顶点 B，但不能直接从 B 到 A，而 B 和 C 之间有两条边分别指向两个方向，因此可以双向移动。

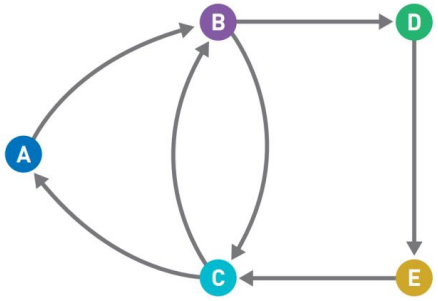


图 71.　　有向图

和无向图一样，有向图的边也可以加上权重。

在下图中，从顶点 B 到顶点 C 的权重为 5，而从 C 到 B 的权重为 7。如果做的是一个表示移动时间的图，而从 B 到 C 是下坡路，就有可能出现这样的情况。

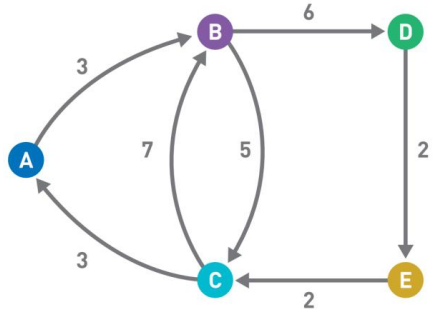


图 72.　　加权有向图

5. 图能给我们带来哪些便利

想一想图能给我们带来的好处吧。假设图中有两个顶点 s 和 t，而我们设计出了一种

算法，可以找到“从 s 到 t 的权重之和最小”的那条路径。

那么，这种算法就可以应用到这些问题上：寻找计算机网络中通信时间最短的路径，寻找路线图中耗时最短的路径，寻找路线图中最省乘车费的路径等 A。

就像这样，只要能用图来表示这些关系，我们就可以用解决图问题的算法来解决这些看似不一样的问题。