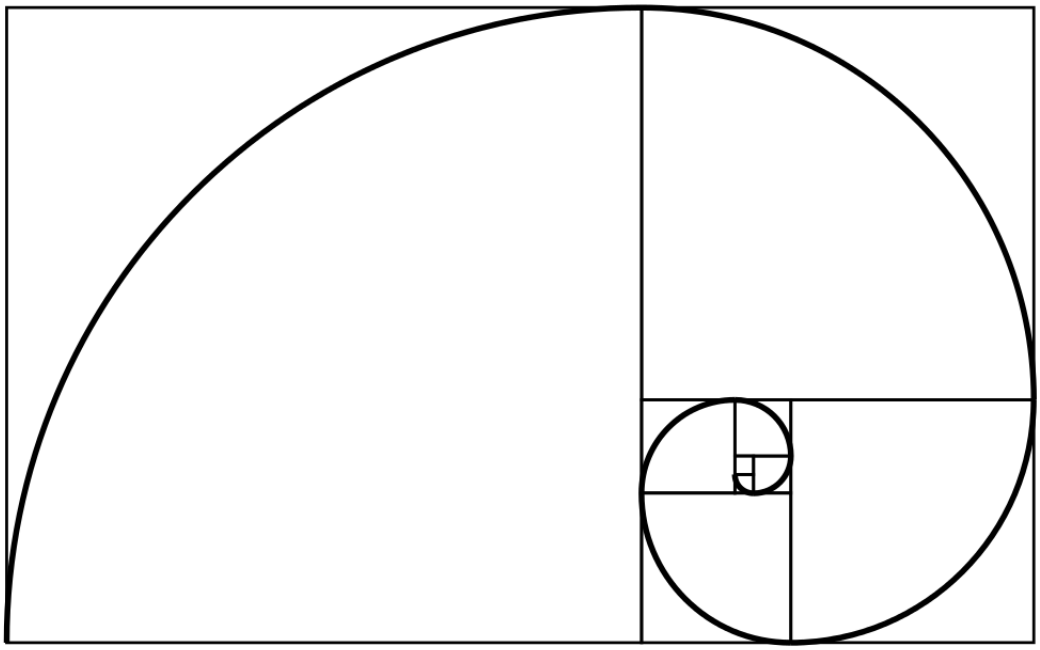
基本算法



刘新宇 [1]

2021 年 4 月 4 日

[1]**刘新宇**
Version: 0.6180339887498949
Email: liuxinyu95@gmail.com

目录

尽管我们在课堂上学习基本算法，但除了编程竞赛，求职面试，很多人在工作中根本用不上。当人们谈到人工智能和机器学习算法时，实际上说的是数学模型而非基本算法和数据结构。即使在工作中遇到算法，大多数时候程序库中已经实现好了。我们只需要了解如何使用，而不用自己重新实现。

算法在解决一些“有趣”的问题时，会扮演关键角色。作为例子，让我们来看看下面这两个趣题。

0.1 最小可用数

理查德 · 伯德提出过一个问题：找出不在一个列表中出现的最小数字（[1] 第一章）。我们经常使用数字作为标识某一实体的标签，例如身份证号，银行账户，电话号码等等。一个数字或者被占用，或者没有被占用。我们希望找到一个最小的没有被占用数字。假设数字都是非负整数，所有正在被使用的数字记录在一个列表中：

```
[18, 4, 8, 9, 16, 1, 14, 7, 19, 3, 0, 5, 2, 11, 6]
```

不在这个列表中的最小整数是 10。这个题目看上去是如此简单，我们可以立即写出下面解法：

```
1: function MIN-FREE(A)
2:     x ← 0
3:     loop
4:         if x ∉ A then
5:             return x
6:         else
7:             x ← x + 1
```

其中符号 $\notin$ 的实现如下：

```
1: function '∉'(x, X)
2:     for i ← 1 to |X| do
3:         if x = X[i] then
4:             return False
5:     return True
```

有些编程语言内置了这一线性查找的实现，下面是一段例子代码。

```
def minfree(lst):
    i = 0
    while True:
        if i not in lst:
            return i
        i = i + 1
```

当列表存储了几百万个数字时，这个方法的的性能很快变差。它消耗的时间和列表长度的平方成正比。在一台双核 2.10GHz 处理器，2G 内存的计算机上，其 C 语

言实现需要 5.4 秒才能在十万个数字中找到答案。当数量上升到一百万时，则耗时达到 8 分钟。

0.1.1 改进

改进这一解法的关键基于这一事实：对于任何 n 个非负整数 $x_1, x_2, ..., x_n$，如果存在小于 n 的可用整数，必然存在某个 x_i 不在 $[0, n)$ 这个范围内。否则这些整数一定是 $0, 1, ..., n-1$ 的某个排列，这种情况下，最小的可用整数是 n。于是我们有如下结论：

$$minfree(x_1, x_2, ..., x_n) \leq n \tag{1}$$

为此我们可以用一个长度为 $n+1$ 的数组，来标记区间 $[0, n]$ 内的某个整数是否可用。

```
1: function MIN-FREE(A)
2:     F ←[False, False, ..., False] where |F| = n + 1
3:     for ∀x ∈ A do
4:         if x < n then
5:             F[x] ← True
6:     for i ← [0, n] do
7:         if F[i] = False then
8:             return i
```

其中第 2 行将标志数组中的所有值初始化为假。接着我们遍历 A 中的所有数字，只要小于 n，就将相应的标记置为真。接下来我们再次扫描标志数组，找到第一个值为假的位置。整个算法用时和 n 成正比。我们使用了 $n+1$ 而不是 n 个标志，这样无需额外处理，就可以应对 $sorted(A) = [0, 1, 2, ..., n-1]$ 的特殊情况。

虽然这个方法只需要线性时间，但是它需要 $O(n)$ 的空间来存储标志。我们还可以继续优化。每次查找都要申请长度为 $n+1$ 的数组；查找结束后，这个数组又被释放了。反复的申请和释放会消耗大量时间。我们可以预先准备好足够长的数组，然后每次查找都复用它。另外，我们可以使用二进制的位来保存标志，从而节约空间。下面的 C 语言例子程序实现了这两点改进：

```
#define N 1000000
#define WORD_LENGTH (sizeof(int) * 8)

void setbit(unsigned int* bits, unsigned int i) {
    bits[i / WORD_LENGTH] |= 1 << (i % WORD_LENGTH);
}

int testbit(unsigned int* bits, unsigned int i) {
    return bits[i / WORD_LENGTH] & (1 << (i % WORD_LENGTH));
}
```

```
unsigned int bits[N / WORD_LENGTH + 1];

int minfree(int* xs, int n) {
  int i, len = N/WORD_LENGTH + 1;
  for (i = 0; i < len; ++i) {
      bits[i]=0;
  }
  for (i=0; i < n; ++i) {
      if(xs[i] < n) {
          setbit(bits, xs[i]);
      }
  }
  for (i=0; i <= n; ++i) {
      if (!testbit(bits, i)) {
          return i;
      }
  }
}
```

在相同的计算机上，这段程序仅用 0.023 秒就可以处理一百万个数字。

0.1.2 分而治之

我们在速度上的改进是以空间上的消耗为代价的。由于维护了一个长度为 n 的标志数组，当 n 很大时，空间就会成为新的瓶颈。分而治之的策略将问题分解为若干规模较小的子问题，然后逐步解决它们以得到最终的结果。

我们可以将所有满足 $x_i \leq \lfloor n/2 \rfloor$ 的整数放入一个子序列 A'；将其它整数放入另外一个序列 A''。根据公式1，如果序列 A' 的长度正好是 $\lfloor n/2 \rfloor$，这说明前一半的整数 A' 已经“满了”，最小可用整数一定可以在 A'' 中找到。否则，最小可用整数一定在 A' 中。总之，通过这一划分，问题的规模减小了。

需要注意的是，当在子序列 A'' 中递归查找时，边界情况发生了一些变化。我们不再是从 0 开始寻找最小可用整数，查找的下界变成了 $\lfloor n/2 \rfloor + 1$。因此我们的算法应定义为 $search(A, l, u)$，其中 l 是下界，u 是上界。递归的边界条件是当序列为空时，我们返回下界 l 作为结果。

$$minfree(A) = search(A, 0, |A| - 1)$$

$$\begin{aligned} search(\varnothing, l, u) &= l \\ search(A, l, u) &= \begin{cases} |A'| = m - l + 1: & search(A'', m + 1, u) \\ otherwise: & search(A', l, m) \end{cases} \end{aligned}$$

其中

$$\begin{aligned} m &= \lfloor \frac{l + u}{2} \rfloor \\ A' &= [x | x \in A, x \leq m] \\ A'' &= [x | x \in A, x > m] \end{aligned}$$

这一方法并不需要额外的空间[1]。每次调用需要进行 $O(|A|)$ 次比较来划分出子序列 A' 和 A''。每次问题的规模都会减半，所以算法用时为 $T(n) = T(n/2) + O(n)$，通过主定理化简得到结果 $O(n)$。我们也可以这样分析：第一次需要 $O(n)$ 次比较来划分子序列 A' 和 A''，第二次需要比较 $O(n/2)$ 次，第三次需要比较 $O(n/4)$ 次……总时间为 $O(n + n/2 + n/4 + ...) = O(2n) = O(n)$。在定义中我们用表达式 $[a|a \in A, p(a)]$ 来定义列表。它和集合表达式 $\{a|a \in A, p(a)\}$ 有所不同。下面的 Haskell 例子代码实现了分而治之的算法。

```
minFree xs = bsearch xs 0 (length xs - 1)

bsearch xs l u | xs == [] = l
               | length as == m - l + 1 = bsearch bs (m+1) u
               | otherwise = bsearch as l m
    where
      m = (l + u) `div` 2
      (as, bs) = partition (≤m) xs
```

0.1.3 简洁与性能

有人会担心这一算法的性能。递归的深度为 $O(\lg n)$，调用栈的大小也是 $O(\lg n)$。我们可以通过将递归转换为迭代来避免空间上的占用：

```
1: function MIN-FREE(A)
2:     l ← 0, u ← |A|
3:     while u − l > 0 do
4:         m ← l + (u − l)/2
5:         left ← l
6:         for right ← l to u − 1 do
7:             if A[right] ≤ m then
8:                 A[left] ↔ A[right]
9:                 left ← left + 1
10:        if left < m + 1 then
11:            u ← left
12:        else
13:            l ← left
```

如图1所示，这段程序对数组中的元素进行划分。$left$ 之前的元素都不大于 m，而 $left$ 和 $right$ 之间的元素都大于 m。

这一解法运行快速并且不需要额外的栈空间。但前面的递归算法更显简洁。不同读者的偏好可能会有所不同。

[1] 递归需要 $O(\lg n)$ 的栈空间，但可以通过尾递归优化消除。

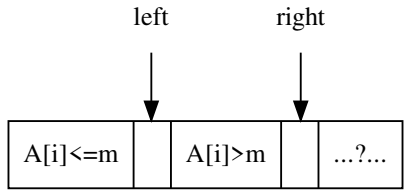


图 1: 数组划分。位于 $0 \le i < left$ 的元素满足 $A[i] \le m$，位于 $left \le i < right$ 的元素满足 $A[i] > m$，剩余的元素尚未处理。

0.2 正规数

第二道趣题是寻找第 1500 个正规数。正规数就是只含有 2、3、5 这三个因子的自然数。因为最大的素因子是 5，所以在数论中又叫作 5-光滑数。在计算机科学中又叫作哈明数以纪念理查德 · 哈明。2、3、5 本身自然也是正规数。$60 = 2^2 3^1 5^1$ 是第 25 个正规数。数字 $21 = 2^0 3^1 7^1$ 由于含有因子 7，所以不是正规数。我们定义 $1 = 2^0 3^0 5^0$ 是第 0 个正规数。前 10 个正规数如下：

1, 2, 3, 4, 5, 6, 8, 9, 10, 12, ...

0.2.1 穷举法

我们可以从 1 开始，逐一检查所有自然数，对于每个整数，把 2、3、5 这些因子不断去掉，然后检查最终结果是否为 1：

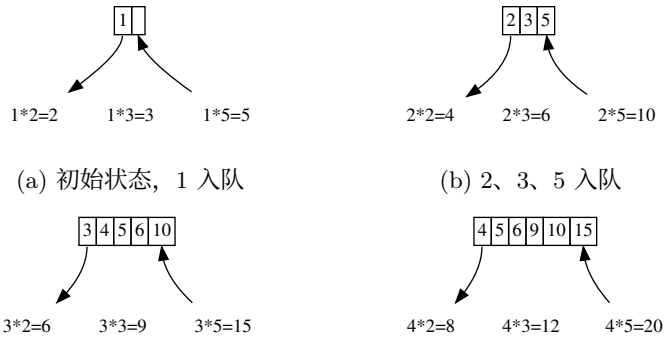
```
1: function REGULAR-NUMBER(n)
2:     x ← 1
3:     while n > 0 do
4:         x ← x + 1
5:         if VALID?(x) then
6:             n ← n − 1
7:     return x

8: function VALID?(x)
9:     while x mod 2 = 0 do
10:        x ← ⌊x/2⌋
11:    while x mod 3 = 0 do
12:        x ← ⌊x/3⌋
13:    while x mod 5 = 0 do
14:        x ← ⌊x/5⌋
15:    return x = 1 ?
```

穷举法对于较小的 n 没有问题。在同样的计算机上，其 C 语言实现用时 40.39 秒才找到第 1500 个正规数（860934420）。当 n 增加到 15000 时，即使 10 分钟也无法结束。

0.2.2 构造性解法

取模和除法是比较耗时 [2]，并且这些运算被循环执行了很多次。我们可以转换思路，不再检查一个数是否仅含 2、3、5 作为因子，而是从这三个因子构造正规数。这样问题就转换为如何从小到大依次产生正规数序列。我们可以使用队列这种数据结构来解决。

队列可以从一侧放入元素，从另一侧取出元素。所以先放入的元素会先被取出。这一特性被称为先进先出 FIFO(First-In-First-Out)。我们的思路是先把 1 作为第 0 个正规数放入队列。然后不断从队列另一侧取出正规数，分别乘以 2、3、5，产生 3 个新正规数，按照大小顺序将其放入队列。如果新产生的数已存在于队列中，则将其丢弃以避免重复。新产生的正规数还可能小于队列尾部的值，因此在插入时，需要保持它们在队列中的大小顺序。图2描述了这一思路的步骤。



(a) 初始状态，1 入队　(b) 2、3、5 入队

(c) 4、6、10 入队　(d) 9、15 入队，6 被丢弃

图 2: 使用队列生成正规数的前 4 步

根据这一思路的算法实现如下：

```
1: function REGULAR-NUMBER(n)
2:     Q ← ∅
3:     x ← 1
4:     ENQUEUE(Q, x)
5:     while n > 0 do
6:         x ← DEQUEUE(Q)
7:         UNIQUE-ENQUEUE(Q, 2x)
8:         UNIQUE-ENQUEUE(Q, 3x)
9:         UNIQUE-ENQUEUE(Q, 5x)
```

```
        n ← n − 1
    return x

function UNIQUE-ENQUEUE(Q, x)
    i ← 0, m ← |Q|
    while i < m and Q[i] < x do
        i ← i + 1
    if i ≥ m or x ≠ Q[i] then
        INSERT(Q, i, x)
```

对于长度为 m 的队列，INSERT 函数需要 $O(m)$ 的时间按序、无重复地插入一个新元素。队列的长度会随着 n 线性增加（每取出一个元素后最多插入三个新元素，增加的比率 ≤ 2），总运行时间为 $O(1+2+3+...+n) = O(n^2)$。

图3的数据显示了队列的访问次数和 n 之间的关系，其形状为二次曲线，反映出 $O(n^2)$ 的复杂度。

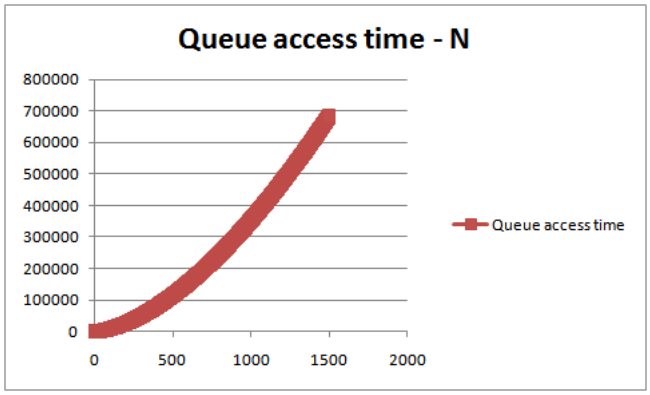


图 3: 队列访问次数和 n 的关系

在同样的计算机上，对应的 C 语言实现仅用 0.016 秒就输出了答案 86093442，比穷举法快 2500 倍。

这一解法也可以用递归的方式给出,令 X 为包含所有正规数的无穷序列 $[x_1, x_2, x_3, ...]$。对于每个正规数，将其乘以 2 得到的仍然是无穷正规数列：$[2x_1, 2x_2, 2x_3, ...]$。同样，依次乘以 3 和 5 会得到另外两个无穷正规数列。如果将这 3 个新无穷数列合并，去除重复元素，然后将 1 添加到最开始，我们就又得到了 X。也就是说，下面的等式成立：

$$X = 1 : [2x|\forall x \in X] \cup [3x|\forall x \in X] \cup [5x|\forall x \in X] \tag{2}$$

其中 $x : X$ 表示将元素 x 连接到列表 X 的前面，从而使 x 成为第一个元素。在 Lisp 中这个操作称为 cons。1 是第 0 个正规数，我们把它放在最前面。为了实现无穷列表的合并，我们递归地定义 $\cup$ 操作。令 $X = [x_1, x_2, x_3...]$，$Y = [y_1, y_2, y_3, ...]$ 为

两个无穷序列。$X' = [x_2, x_3, ...]$，$Y' = [y_2, y_3, ...]$ 表示除去第一个元素后的剩余部分，定义 $\cup$ 为：

$$X \cup Y = \begin{cases} x_1 < y_1 : & x_1 : X' \cup Y \\ x_1 = y_1 : & x_1 : X' \cup Y' \\ y_1 < x_1 : & y_1 : X \cup Y' \end{cases}$$

因为是无穷序列，我们无需处理 X、Y 为空的情况。在支持惰性求值的环境中，算法可以实现为如下的例子代码：

```
ns = 1 : (map (*2) ns) `merge` (map (*3) ns) `merge` (map (*5) ns)

merge (x:xs) (y:ys) | x < y = x : merge xs (y:ys)
                    | x == y = x : merge xs ys
                    | otherwise = y : merge (x:xs) ys
```

通过 `ns !! 1500`，可以得到第 1500 个正规数。在同样的计算机上，这一程序用时 0.03 秒。

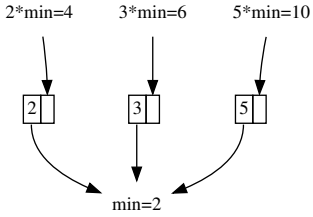
0.2.3 队列

上面的解法虽然快了很多，但会产生重复的元素。它们最终被丢弃了。它需要扫描队列以保证元素有序。入队操作从常数时间退化为线性时间 $O(|Q|)$。为了避免重复，我们把所有正规数分成三个类：$Q_2 = \{2^i | i > 0\}$ 仅包含被 2 整除的数；$Q_{23} = \{2^i 3^j | i \geq 0, j > 0\}$；$Q_{235} = \{2^i 3^j 5^k | i, j \geq 0, k > 0\}$。其中 Q_{23} 要求 $j \neq 0$，Q_{235} 要求 $k \neq 0$。这保证了三类数彼此间没有重复。每类数我们都用一个队列来产生。它们初始化为 $Q_2 = \{2\}$，$Q_{23} = \{3\}$ 和 $Q_{235} = \{5\}$。每次从这三个队列的头部选出最小的元素 x 并取出，然后进行下面的检查：

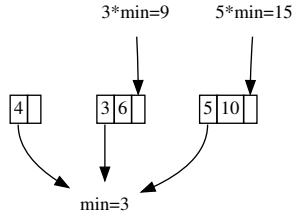
- 如果 x 是从 Q_2 取出的，我们将 $2x$ 加入 Q_2，$3x$ 加入 Q_{23}，$5x$ 加入 Q_{235}。
- 如果 x 是从 Q_{23} 取出的，我们只将 $3x$ 加入 Q_{23}，$5x$ 加入 Q_{235}。我们不应该将 $2x$ 加入 Q_2，因为 Q_3 中不允许包含被 3 整除的数。
- 如果 x 是从 Q_{235} 取出的，我们只将 $5x$ 加入 Q_{235}。我们不应该将 $2x$ 加入 Q_2，$3x$ 加入 Q_{23}，因为它们不允许包含被 5 整除的数。

我们不断从这三个队列中取出最小的，直到取出第 n 个元素。图4给出了前 4 步。按照这个思路，算法可以实现如下。

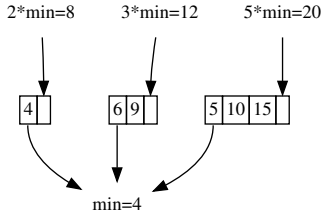
```
1: function REGULAR-NUMBER(n)
2:     x ← 1
3:     Q2 ← {2}, Q23 ← {3}, Q235 ← {5}
4:     while n > 0 do
5:         x ← min(HEAD(Q2), HEAD(Q23), HEAD(Q235))
```



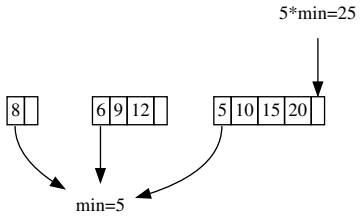
(a) 4、6、10 入队



(b) 9、15 入队



(c) 8、12、20 入队



(d) 25 入队

图 4: 使用三个队列 Q_2、Q_{23} 和 Q_{235} 来构造正规数的前 4 步。初始时它们包含 2、3、5 为唯一元素

```
        if x = HEAD(Q_2) then
            DEQUEUE(Q_2)
            ENQUEUE(Q_2, 2x)
            ENQUEUE(Q_23, 3x)
            ENQUEUE(Q_235, 5x)
        else if x = HEAD(Q_23) then
            DEQUEUE(Q_23)
            ENQUEUE(Q_23, 3x)
            ENQUEUE(Q_235, 5x)
        else
            DEQUEUE(Q_235)
            ENQUEUE(Q_235, 5x)
        n ← n − 1
    return x
```

算法循环 n 次。每次循环，它从三个队列中取出最小的一个元素，这一步需要常数时间。接着它根据取出元素所在的队列，产生一到三个新元素放入队列，这一步也是常数时间。因此整个算法的复杂度是 $O(n)$ 的。

0.3 小结

两个趣题表面上看来都能用简单的穷举法解决。但随着问题规模的增加，我们不得不寻求更好的解法。很多以前难以解决的问题，我们可以通过编程用新的方式找到答案。本书介绍常见的基本算法和数据结构，同时给出函数式和命令式的对比实现。主要参考了冈崎的著作 [3] 和经典的算法教材 [4]。本书尽量避免依赖于特定的编程语言。一方面读者会有自己的语言偏好，另一方面编程语言也在不断变化。为此我们主要使用伪代码和数学记法对算法进行定义，而附以一些例子代码片段。函数式的代码例子类似 Haskell，命令式的代码例子是一些常见语言的混合。这些示例并不一定严格遵循某些语言规范。

本书中文版《算法新解》于 2017 年出版。2020 年底开始进行重写。电子版可以在 github 上获得，如果希望获得纸质版，请联系 liuxinyu95@gmail.com。

练习 1

1. 最小可用数趣题中，所有数都是非负整数。我们可以利用正负号来标记一个数字是否存在。首先扫描一遍列表，令列表长度为 n，对于任何绝对值小于 n 的数 $|x| < n$，将位置 $|x|$ 上的数字置为负数。之后再次扫描一遍列表，找到第一个整数所在的位置就是答案。编程实现这一算法。
2. n 个数字 1, 2, ..., n，经过某一处理后，它们的顺序被打乱了，并且某一个数

x 被改成了 y。假设 $1 \leq y \leq n$，设计一个方法能够在线性时间、常数空间内找出 x 和 y。

3. 下面是一段求正规数的代码。它是一种队列解法么？

```
Int regularNum(Int m) {
    nums = Int[m + 1]
    n = 0, i = 0, j = 0, k = 0
    nums[0] = 1
    x2 = 2 * nums[i]
    x3 = 3 * nums[j]
    x5 = 5 * nums[k]
    while (n < m) {
        n = n + 1
        nums[n] = min(x2, x3, x5)
        if (x2 == nums[n]) {
            i = i + 1
            x2 = 2 * nums[i]
        }
        if (x3 == nums[n]) {
            j = j + 1
            x3 = 3 * nums[j]
        }
        if (x5 == nums[n]) {
            k = k + 1
            x5 = 5 * nums[k]
        }
    }
    return nums[m];
}
```

第一章　列表

1.1　简介

列表和数组是构建其它复杂数据结构的基石。它们都可以看作是容纳若干元素的容器。数组通常是一组连续的存储区域，每个存储单元由一个数字索引。这个数字叫作地址或者位置。数组的大小是有限的，通常需要在使用前确定。与数组不同，列表的大小无需预先确定，可以随时加入新元素。我们可以从头到尾依次遍历列表中的元素。特别是在函数式环境中，列表相关算法对于计算和逻辑的控制起着关键作用[1]。对于已经熟悉映射 (map)，过滤 (filter)，叠加 (fold) 等算法的读者，可以跳过这一章，直接从第二章开始阅读。

1.2　定义

列表又称单向链表，是一种递归的数据结构。其定义如下：

- 一个**列表**或者为空，记为 ∅ 或 NIL；
- 或者包含一个元素和一个**列表**。

图1.1描述了一个由若干节点组成的列表。每个节点包含两部分，一个元素（也称作 key）和一个子列表。指向子列表的引用通常叫作 next。最后一个节点中的子列表为空，记为‘NIL’。

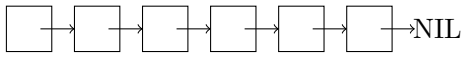


图 1.1: 由节点组成的列表

每个节点要么链接到下一个节点上，要么指向 NIL。通常使用复合数据结构[2]定义列表，例如：

```
struct List<A> {
    A key
```

[1]在更底层，lambda 演算作为和图灵机等价的计算模型更为基础 [93], [99]。

[2]多数情况下，列表中元素有着共同的类型。有些环境（如 Lisp）支持包含不同数据类型的列表。

```
    List<A> next
}
```

这里需要对“空”列表的概念加以说明。很多传统的编程环境支持空引用 null 概念，因此存在两种不同的方法表示空列表。一种直接使用空引用 null（或 NIL）；另一种创建一个列表，但不填入任何元素，通常表示为 []。在实现上，空引用无需占用内存，但 [] 则需要分配内存。本书使用符号 $\varnothing$ 表示抽象的空列表、空集、空容器。

1.2.1 分解

给定一个非空列表 L，我们定义两个函数来分别获取头部元素和子列表。它们通常被命名为 $first(L)$ 和 $rest(L)$，或者 $head(L)$ 和 $tail(L)$[3]。反之，我们可以从一个元素 x 和列表 xs（可为空）构造出另一个列表，记为 $x : xs$。这一构造过程也叫作 cons。我们有如下关系：

$$\begin{cases} head(x : xs) & = x \\ tail(x : xs) & = xs \end{cases} \tag{1.1}$$

对于非空列表 X，我们也用 x_1 表示第一个元素，用 X' 表示剩余列表，例如 $X = [x_1, x_2, x_3, ...]$，$X' = [x_2, x_3, ...]$。

练习 1.2

1. 对于元素类型为 A 的列表，如果能够判断任何两个元素 $x, y \in A$ 是否相等，定义一个算法来判断两个列表是否相等。

1.2.2 列表的基本操作

根据定义，我们可以递归地计算列表的长度：空列表的长度为 0，而非空列表的长度是除去第一个元素的子列表长度加一。

$$\begin{aligned} length(\varnothing) &= 0 \\ length(L) &= 1 + length(L') \end{aligned} \tag{1.2}$$

为了计算长度，我们从头到尾遍历列表。其时间复杂度是 $O(n)$，其中 n 是元素个数。为了避免反复计数，我们可以将长度存储在一个变量中，并在增加或删除元素时更新这一变量。下面是计算长度的迭代实现：

```
1: function LENGTH(L)
2:     n ← 0
3:     while L ≠ NIL do
4:         n ← n + 1
```

[3]在 Lisp 中，由于历史原因，它们被命名为 car 和 cdr 用以代表当时机器中的寄存器 [63]

```
5:        L ← NEXT(L)
6:    return n
```

在不和绝对值混淆的情况下，我们也使用 $|L|$ 来表示列表 L 的长度。

1.2.3 索引

数组支持以常数时间随机访问任意位置 i 的元素。但列表需要前进 i 步才能到达元素所在位置。

$$getAt(i,\ x : xs) = \begin{cases} i = 0 : & x \\ i \neq 0 : & getAt(i-1, xs) \end{cases} \tag{1.3}$$

为了从一个非空列表中获取第 i 个元素：

- 若 i 为 0，结果为列表中的头部元素；
- 否则，结果为子列表中的第 $i-1$ 个元素。

我们故意没有处理空列表的情况。如果传入 $\varnothing$，此时的行为是未定义的。i 越界时的行为也是未定义的。若 $i > |L|$，通过递归，最终转化为访问空列表的第 $i - |L|$ 个位置的情况。另一方面，若 $i < 0$，继续减一将使得它更偏离 0，最终转化为访问空列表的某个负索引位置的情况。

由于需要前进 i 步，索引算法的时间复杂度为 $O(i)$。下面是对应的迭代实现：

```
1: function GET-AT(i, L)
2:     while i ≠ 0 do
3:         L ← NEXT(L)                          ▷ L = NIL 时出错
4:         i ← i − 1
5:     return FIRST(L)
```

练习 1.3

1. 在 GET-AT(i, L) 的迭代实现中，L 为空会怎样？i 越界时会怎样？

1.2.4 末尾元素

存在一对和 first/rest 对称的操作,称为 last/init。对于非空列表 $X = [x_1, x_2, ..., x_n]$，函数 $last$ 返回末尾元素 x_n，而 $init$ 返回子列表 $[x_1, x_2, ..., x_{n-1}]$。虽然这两对操作左右对称，但 last/init 需要遍历列表，因而是线性时间的。

当获取列表 X 的末尾元素时：

- 如果列表只含有一个元素 $[x_1]$，则 x_1 就是末尾元素；
- 否则，结果为子列表 X' 的末尾元素。

$$\begin{aligned} last([x]) &= x \\ last(x : xs) &= last(xs) \end{aligned} \tag{1.4}$$

类似地，当获取除去末尾元素的子列表时：

- 如果列表只含有一个元素 $[x_1]$，结果为空 $[\,]$；
- 否则，我们递归地从子列表 X' 中获取除去末尾元素的剩余部分，然后将 x_1 附加在前面。

$$\begin{aligned} init([x]) &= [\,] \\ init(x : xs) &= x : init(xs) \end{aligned} \tag{1.5}$$

这两个算法中都没有处理空列表的情况，当传入 $\varnothing$ 时，其行为是未定义的。下面是相应的迭代实现。

```
1: function LAST(L)
2:     x ← NIL
3:     while L ≠ NIL do
4:         x ← FIRST(L)
5:         L ← REST(L)
6:     return x

7: function INIT(L)
8:     L' ← NIL
9:     while REST(L) ≠ NIL do                ▷ L 为 NIL 时出错
10:        L' ← CONS(FIRST(L), L')
11:        L ← REST(L)
12:    return REVERSE(L')
```

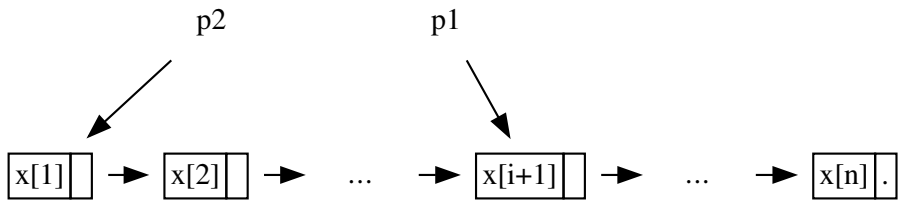
这一算法一边向尾部前进，一边通过 cons 累积 init 的结果。但是这样产生的列表是逆序的，因此最后需要将结果倒转过来（见第1.3.2节）。

1.2.5 反向索引

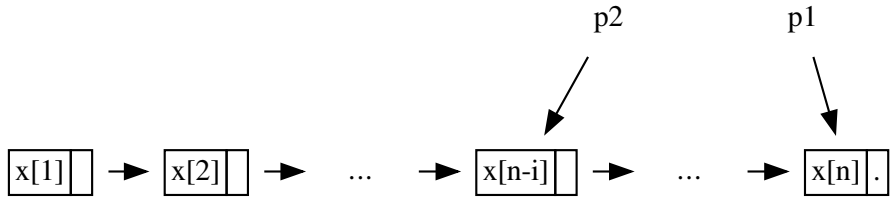
$last()$ 是反向索引的一种特例。更一般的形式是获取列表中的倒数第 i 个元素。最直接的思路是遍历两次：第一次获取列表长度 n，第二次获取第 $n - i - 1$ 个元素：

$$lastAt(i, L) = getAt(|L| - i - 1, L) \tag{1.6}$$

更好的解法是使用两个指针 p_1 和 p_2，它们相距 i 步，即 $rest^i(p_2) = p_1$，其中 $rest^i(p_2)$ 表示重复执行函数 $rest()$ 总共 i 次。也就是说，从 p_2 前进 i 步就可到达 p_1。p_2 一开始指向链表的头部，然后同时向前移动它们，直到 p_1 到达链表的尾部。此时指针 p_2 恰好指向倒数第 i 个元素。图1.2描述了这一方法。由于 p_1, p_2 框出一个窗口，这一方法也称作滑动窗口法。



(a) p_2 开始时指向表头，它在指针 p_1 之后，距离 i 步。



(b) 当 p_1 到达表尾时，p_2 恰好指向从右数第 i 个元素。

图 1.2: 双指针框出一个滑动窗口

```
1: function LAST-AT(i, L)
2:     p ← L
3:     while i > 0 do
4:         L ← REST(L)                          ▷ 越界时出错
5:         i ← i − 1
6:     while REST(L) ≠ NIL do
7:         L ← REST(L)
8:         p ← REST(p)
9:     return FIRST(p)
```

纯函数实现时不能直接更新指针，为此我们可以同时遍历 $X = [x_1, x_2, ..., x_n]$ 和 $Y = [x_i, x_{i+1}, ..., x_n]$，其中 Y 是除去前 $i-1$ 个元素后的子列表。

- 如果 Y 中仅含有一个元素 $[x_n]$，则倒数第 i 个元素就是表头 x_n；
- 否则，我们同时从 X 和 Y 中各丢弃一个元素，然后递归地检查列表 X' 和 Y'。

$$lastAt(i, X) = slide(X, drop(i, X)) \tag{1.7}$$

其中数函 $slide(X, Y)$ 同时丢弃两个列表的头部：

$$\begin{aligned} slide(x : xs,\ [y]) &= x \\ slide(x : xs,\ y : ys) &= slide(xs, ys) \end{aligned} \tag{1.8}$$

函数 $drop(m, X)$ 丢弃列表 X 中的前 m 个元素，我们可以通过前进 m 步实现：

$$\begin{array}{rcl} drop(0,\ X) & = & X \\ drop(m,\ \varnothing) & = & \varnothing \\ drop(m,\ x : xs) & = & drop(m-1, xs) \end{array} \tag{1.9}$$

练习 1.4

1. 在 INIT 算法中，我们可以用 APPEND(L', FIRST(L)) 来替换 cons 么？
2. 在 LAST-AT 算法中，如何处理空列表和越界的情况？

1.2.6 更改

更改操作包括添加、插入、更新、删除。某些函数式环境在实现时创建新列表，而原列表保持 (persist) 不变，并在适当的时候释放原始列表（[3]，第 2 章）。

添加

添加称为 append，它和 cons 对称。一个在表头增加，一个在末尾增加。因此添加也被称作 snoc（将 cons 反过来拼写）。由于要遍历到列表尾部，所以其复杂度为 $O(n)$，其中 n 是列表的长度。为了避免反复遍历，我们可以将尾部位置存储下来，并随着列表变化进行更新。

$$\begin{array}{rcl} append(\varnothing, x) & = & [x] \\ append(y : ys, x) & = & y : append(ys, x) \end{array} \tag{1.10}$$

- 向空列表添加 x，结果为 $[x]$；
- 否则，将 x 添加到子列表的末尾。

对应的迭代实现如下：

```
1: function APPEND(L, x)
2:     if L = NIL then
3:         return CONS(x, NIL)
4:     H ← L                                   ▷ 保存表头
5:     while REST(L) ≠ NIL do
6:         L ← REST(L)
7:     REST(L) ← CONS(x, NIL)
8:     return H
```

更新 REST 的过程通常实现为对 `next` 引用的改写，如下面的例子代码：

```
List<A> append(List<A> xs, T x) {
    if (xs == null) {
        return cons(x, null)
    }
    List<A> head = xs
    while (xs.next ≠ null) {
        xs = xs.next
    }
    xs.next = cons(x, null)
    return head
}
```

练习 1.5

1. 在列表的定义中增加一个尾部变量 tail，将添加算法优化为常数时间。
2. 何时应该更新 tail 变量？对性能有何影响？

修改

和 $getAt$ 类似,我们需要移动到列表中的指定位置以修改元素。定义函数 $setAt(i, x, L)$ 为:

- 若 $i = 0$，要修改的是头部元素，结果为 $x : L'$;
- 否则，递归地修改子列表 L' 中的第 $i - 1$ 个元素。

$$\begin{aligned} setAt(0, x,\ y : ys) &= x : ys \\ setAt(i, x,\ y : ys) &= y : setAt(i - 1, x, ys) \end{aligned} \tag{1.11}$$

这一算法的时间复杂度为 $O(i)$，其中 i 是要修改的位置。

练习 1.6

1. 在 $setAt$ 中，如何处理空列表和越界的情况？

插入

列表插入有两种不同的含义:一个是在指定位置插入一个元素,记为 $insert(i, x, L)$,其实现和 $setAt$ 类似；另一含义是在已序列表中插入一个元素，使得结果仍然是已序的。

为了插入元素 x，需要先前进 i 步到达插入位置。然后用 x 和后续子列表构造一个新列表，再和前 i 个元素链接起来[4]。

- 若 $i = 0$，插入就转变成了 cons，结果为 $x : L$;

[4] i 从 0 开始。

- 否则，递归地将 x 插入到子列表 L' 的第 $i-1$ 个位置，并将原头部元素附加在前面。

$$\begin{aligned} insert(0, x,\ L) &= x : L \\ insert(i, x,\ y : ys) &= x : insert(i-1, x, ys) \end{aligned} \tag{1.12}$$

当 i 超过列表的长度时，我们可以将其视作添加，见本节习题。下面是相应的迭代实现：

```
1: function INSERT(i, x, L)
2:     if i = 0 then
3:         return CONS(x, L)
4:     H ← L
5:     p ← L
6:     while i > 0 and L ≠ NIL do
7:         p ← L
8:         L ← REST(L)
9:         i ← i − 1
10:    REST(p) ← CONS(x, L)
11:    return H
```

如果列表 $L = [x_1, x_2, ..., x_n]$ 已序，即对任何位置 $1 \le i \le j \le n$，有 $x_i \le x_j$。这里 $\le$ 的含义是抽象的，它可以代表任何有序的比较，包括 $\ge$（降序）、集合的包含关系等。我们可以设计一个算法，使得新元素 x 插入 L 后列表仍然有序。

- 若 L 为空或者 x 小于 L 的头部元素，结果为 $x : L$；
- 否则，我们递归地将元素 x 插入到子列表 L' 中。

$$\begin{aligned} insert(x,\ \varnothing) &= [x] \\ insert(x,\ y : ys) &= \begin{cases} x \le y: & x : y : ys \\ \text{否则}: & y : insert(x, ys) \end{cases} \end{aligned} \tag{1.13}$$

由于要逐一比较元素，插入的时间复杂度为 $O(n)$，其中 n 是长度。对应的迭代实现如下：

```
1: function INSERT(x, L)
2:     if L = NIL or x < FIRST(L) then
3:         return CONS(x, L)
4:     H ← L
5:     while REST(L) ≠ NIL and FIRST(REST(L)) < x do
6:         L ← REST(L)
7:     REST(L) ← CONS(x, REST(L))
```

```
8:    return H
```

利用按序插入操作，我们可以实现插入排序：逐一将元素按序插入到一个空列表中。由于每次按序插入都是线性的，所以这一排序的复杂度为 $O(n^2)$。

$$\begin{aligned} sort(\varnothing) &= \varnothing \\ sort(x : xs) &= insert(x, sort(xs)) \end{aligned} \tag{1.14}$$

这是一个递归算法：先递归地将子列表排序，然后把第一个元素按序插入。我们可以消除递归，实现一个迭代算法：逐一从列表中取出元素并按序插入到结果中：

```
1: function SORT(L)
2:     S ← NIL
3:     while L ≠ NIL do
4:         S ← INSERT(FIRST(L), S)
5:         L ← REST(L)
6:     return S
```

在循环中的任何时刻，结果列表都是已序的。和递归实现相比，它们有一个本质不同：前者从右向左处理列表，而后者从左向右处理。我们稍后将在“尾递归” 1.2.7节中讲述如何消除这一差异。第 3 章详细介绍插入排序，包括性能分析和优化。

练习 1.7

1. 当插入位置越界时，将其按照添加来处理。
2. 针对数组实现插入算法，插入位置 i 后的所有元素需要向后移动一个位置。
3. 只使用小于 $<$ 比较实现插入排序。

删除

和插入类似，删除也有两种含义：一种是在指定位置删除元素；另一种是查找某个值并删除。前者定义为 $delAt(i, L)$，后者定义为 $delete(x, L)$。

为了删除位置 i 上的元素，我们首先前进 i 步到达目标位置，然后跳过一个元素，将剩余部分连接起来。

- 若列表 L 为空，则结果为空列表；
- 若 $i = 0$，要删除的是头部元素，结果为 L'；
- 否则，递归地从子列表 L' 中删除第 $i - 1$ 个元素，然后将原列表头部附加在前。

$$\begin{aligned} delAt(i,\ \varnothing) &= \varnothing \\ delAt(0,\ x : xs) &= xs \\ delAt(i,\ x : xs) &= x : delAt(i - 1, xs) \end{aligned} \tag{1.15}$$

由于需要前进 i 步执行删除，这一算法的时间复杂度为 $O(i)$。下面是相应的迭代实现：

```
1: function DEL-AT(i, L)
2:     S ← CONS(⊥, L)                                  ▷ 辅助节点
3:     p ← S
4:     while i > 0 and L ≠ NIL do
5:         i ← i − 1
6:         p ← L
7:         L ← REST(L)
8:     if L ≠ NIL then
9:         REST(p) ← REST(L)
10:    return REST(S)
```

为了简化边界情况的处理，我们引入来一个辅助节点 S，它包含一个特殊的值 $\perp$，并指向 L。使用 S，我们可以安全地切除 L 中的任何节点，包括头节点。最后，我们将 S 后继的部分作为结果返回，并丢弃 S 自身。

“查找并删除”又可以进一步细分为两种情况：一种是仅找到第一个出现的元素并删除；另外一种是找到所有等于指定值的元素全部删除。后者更加一般，见本节练习。当在列表 L 中删除 x 时：

- 如果列表为空，则结果为 $\varnothing$；
- 否则，比较表头和 x，若相等，则结果为 L'；
- 若表头不等于 x，则保留表头，并递归地在 L' 中删除 x。

$$\begin{aligned} delete(x,\ \varnothing) &= \varnothing \\ delete(x,\ y:ys) &= \begin{cases} x = y: & ys \\ x \neq y: & y : delete(x, ys) \end{cases} \end{aligned} \tag{1.16}$$

由于需要遍历列表以查找待删除的元素，这一算法的复杂度为 $O(n)$，其中 n 为长度。在迭代实现中，我们依然可以使用辅助节点来简化逻辑：

```
1: function DELETE(x, L)
2:     S ← CONS(⊥, L)
3:     p ← L
4:     while L ≠ NIL and FIRST(L) ≠ x do
5:         p ← L
6:         L ← REST(L)
7:     if L ≠ NIL then
8:         REST(p) ← REST(L)
```

```
    return REST(S)
```

练习 1.8

1. 设计算法将等于给定值的所有元素删除。
2. 设计数组的删除算法，被删除位置后的所有元素需要向前移动一个位置。

连接

连接是添加操作的更一般形式，添加每次向列表尾部加入一个元素，而连接向列表尾部加入多个元素。但如果通过多次添加来实现，则整体操作的性能会下降为平方级别：

$$\begin{aligned} X \mathbin{+\!\!+} \varnothing &= X \\ X \mathbin{+\!\!+} (y : ys) &= append(X, y) \mathbin{+\!\!+} ys \end{aligned} \tag{1.17}$$

这个实现在连接 X 和 Y 时，每次添加都需要前进到尾部，总共添加 $|Y|$ 次。总时间复杂度为 $O(|X| + (|X| + 1) + ... + (|X| + |Y|)) = O(|X||Y| + |Y|^2)$。考虑链接操作 cons 的速度很快（常数时间），我们可以前进到 X 的尾部，然后链接到 Y：

- 若 X 为空，结果为 Y；
- 否则，我们将子列表 X' 和 Y 连接起来，再把 x_1 附加到头部。

另外，当 Y 为空时，我们无需遍历，可以直接返回 X 作为结果：

$$\begin{aligned} \varnothing \mathbin{+\!\!+} Y &= Y \\ X \mathbin{+\!\!+} \varnothing &= X \\ (x : xs) \mathbin{+\!\!+} Y &= x : (xs \mathbin{+\!\!+} Y) \end{aligned} \tag{1.18}$$

改进的算法只遍历一次 X，然后将其尾部链接到 Y，复杂度为 $O(|X|)$。在命令式环境中，通过使用尾部引用，可以实现常数时间的连接操作（见本节习题）。下面的迭代实现并未使用尾部引用：

```
1: function CONCAT(X, Y)
2:     if X = NIL then
3:         return Y
4:     if Y = NIL then
5:         return X
6:     H ← X
7:     while REST(X) ≠ NIL do
8:         X ← REST(X)
9:     REST(X) ← Y
10:    return H
```

1.2.7 和与积

我们常常需要计算列表中数字的和与积。它们有着共同的计算结构，在第1.5节中，我们介绍如何对它们进行抽象。

递归求和与求积

为了计算列表中元素的和：

- 若列表为空，则结果为 0；
- 否则，结果为第一个元素加上剩余元素的和。

$$\begin{aligned} sum(\varnothing) &= 0 \\ sum(x : xs) &= x + sum(xs) \end{aligned} \tag{1.19}$$

求积时，不能简单地将加法替换为乘法，否则结果总为 0。我们需要定义空列表的积为 1。

$$\begin{aligned} product(\varnothing) &= 1 \\ product(x : xs) &= x \cdot product(xs) \end{aligned} \tag{1.20}$$

两个算法都需要遍历整个列表，它们的性能为 $O(n)$，其中 n 为长度。

尾递归

求和、求积算法都从右向左计算。我们可以将其改成从左向右**累积计算**。求和时，结果从 0 开始累积，逐一将元素加到结果上。求积时，从 1 开始累积，逐一将元素乘到结果上。累积过程定义如下：

- 若列表为空，返回当前累积结果；
- 否则，取出表头元素，将其累积到结果上，然后继续处理剩余列表。

下面是求和、求积的累积计算：

$$\begin{aligned} sum'(A,\ \varnothing) &= A & \qquad prod'(A,\ \varnothing) &= A \\ sum'(A,\ x : xs) &= sum(x + A, xs) & \qquad prod'(A,\ x : xs) &= prod'(x \cdot A, xs) \end{aligned} \tag{1.21}$$

给定数字列表，我们以 0 为累积起始值调用 sum'，以 1 为累积起始值调用 $prod'$：

$$sum(X) = sum'(0, X) \qquad product(X) = prod'(1, X) \tag{1.22}$$

或使用柯里化形式：

$$sum = sum'(0) \qquad product = prod'(1)$$

柯里化是由肖芬格尔 (Schönfinkel, 1889 - 1942) 在 1924 年提出，后来经哈斯克尔·柯里在 1958 年后被广泛使用的 [73]。考虑二元函数 $f(x, y)$，如果只传入一个参数 x，它就转换为一个关于 y 的一元函数：$g(y) = f(x, y)$，记为 $g = f\ x$。推广到多元函数 $f(x, y, ..., z)$，通过依次传入参数，可以转换为一系列函数：$f, f\ x, f\ x\ y, ...$。我们称这样的转换为柯里化。它可以把多元函数转化为一系列一元函数，即：$f(x, y, ..., z) = f(x)(y)...(z) = f\ x\ y\ ...\ z$。

采用累积法后，不仅计算顺序变为从左向右，并且无需记录任何中间结果或者状态用于递归。所有的状态或作为参数（例如 A）传入，或丢弃不用（例如已处理过的元素）。这样的递归可进一步优化为循环。我们称这样的函数为“尾递归”（或“尾调用”），称这种消除递归的优化为“尾递归优化”[61]。顾名思义，在这类函数中，递归发生在计算的尾部。尾递归优化可以极大地提高性能，并避免由于递归过深造成的调用栈溢出。

在第1.2.6节关于插入排序的部分，其递归实现从右向左对元素排序。我们也可以将其优化为尾递归：

$$\begin{aligned} sort'(A,\ \varnothing) &= A \\ sort'(A,\ x : xs) &= sort'(insert(x, A), xs) \end{aligned} \tag{1.23}$$

这样排序可以定义为传入 $\varnothing$ 作为起始值的柯里化形式：

$$sort = sort'(\varnothing) \tag{1.24}$$

作为尾递归的典型例子，我们考虑如何高效地计算幂 b^n？（[63]，1.16 节。）最直接的方法是从 1 开始重复乘以 b 共 n 次，这是一个 $O(n)$ 时间的方法：

```
1: function POW(b, n)
2:     x ← 1
3:     loop n times
4:         x ← x · b
5:     return x
```

考虑计算 b^8 的过程，上述算法经过前两次迭代，可以得到 $x = b^2$ 的结果。此时，我们无需用 x 乘以 b 得到 b^3，可以直接再次乘以 b^2，从而得到 b^4。然后再次乘方，就可以得到 $(b^4)^2 = b^8$。这样总共只要循环 3 次，而不是 8 次。若 n 恰好为 2 的整数次幂，即 $n = 2^m$，其中 m 是非负整数，我们可以用下面的方法快速计算 b^n：

$$\begin{aligned} b^1 &= b \\ b^n &= (b^{\frac{n}{2}})^2 \end{aligned}$$

继续这一分而治之的想法，我们可以将 n 推广到任意的非负整数：

- 若 $n = 0$，定义 $b^0 = 1$；
- 若 n 为偶数，将 n 减半，先计算 $b^{\frac{n}{2}}$，然后再将结果平方；

- 若 n 为奇数，因为 $n-1$ 是偶数，可以先递归计算 b^{n-1}，然后再将结果乘以 b。

$$\begin{aligned} b^0 &= 1 \\ b^n &= \begin{cases} 2|n: & (b^{\frac{n}{2}})^2 \\ \text{否则}: & b \cdot b^{n-1} \end{cases} \end{aligned} \tag{1.25}$$

但是，第二条调用无法直接转换为尾递归。为此，我们可以先将底数平方，然后再将指数减半。

$$\begin{aligned} b^0 &= 1 \\ b^n &= \begin{cases} 2|n: & (b^2)^{\frac{n}{2}} \\ \text{否则}: & b \cdot b^{n-1} \end{cases} \end{aligned} \tag{1.26}$$

经过这一修改，就可以将算法转换为尾递归。我们通过等式 $b^n = pow(b, n, 1)$ 计算幂。

$$\begin{aligned} pow(b, 0, A) &= A \\ pow(b, n, A) &= \begin{cases} 2|n: & pow(b^2, \frac{n}{2}, A) \\ \text{否则}: & pow(b, n-1, b \cdot A) \end{cases} \end{aligned} \tag{1.27}$$

和最初的方法相比，其性能提高到了 $O(\lg n)$。我们还可以继续改进，如果将 n 表示成二进制数 $n = (a_m a_{m-1} ... a_1 a_0)_2$，如果 $a_i = 1$，我们清楚地知道，需要计算 b^{2^i}。这和二项式堆的情况很类似（节 10.2）。将所有二进制位为 1 的幂计算出，再累乘到一起就得到最后的结果。

例如，当计算 b^{11} 时，11 写成二进制为 $11 = (1011)_2 = 2^3 + 2 + 1$，因此 $b^{11} = b^{2^3} \times b^2 \times b$。我们可以通过以下的步骤进行计算：

1. 计算 b^1，得 b；
2. 从这一结果进而得到 b^2；
3. 将第 2 步的结果平方，从而得到 b^{2^2}；
4. 将第 3 步的结果平方，得到 b^{2^3}。

最后，我们将第 1、2、和第 4 步的结果乘到一起，得到 b^{11}。综上，我们可以进一步将算法改进如下。

$$\begin{aligned} pow(b, 0, A) &= A \\ pow(b, n, A) &= \begin{cases} 2|n: & pow(b^2, \frac{n}{2}, A) \\ \text{否则}: & pow(b^2, \lfloor \frac{n}{2} \rfloor, b \cdot A) \end{cases} \end{aligned} \tag{1.28}$$

这一算法本质上每次将 n 向右移动一个二进制位（通过将 n 除以 2）。若 LSB(Least Significant Bit，即最低位) 为 0，n 为偶数。我们将底数平方，继续递

归，无需改变累积结果 A。这对应上面例子的第 3 步；若 LSB 为 1，n 为奇数。除了将底数平方，还要将 b 乘到累积结果 A 上；当 n 为 0 时，我们已处理完 n 中的所有位，最终结果就是累积值 A。在任何时候，更新的底数 b'，移位后的指数 n'，和累积结果 A 总满足不变条件 $b^n = A \cdot (b')^{n'}$。

此前的算法当 n 为奇数时，仅仅将其减一转化为偶数进行处理；这一改进每次都将 n 减半。若 n 的二进制表示中有 m 位，这一算法只计算 m 轮。当然它的复杂度仍然为 $O(\lg n)$。我们将这一算法的命令式实现作为本节练习。

回到求和、求积问题。在迭代实现中，我们一边遍历，一边应用加法或乘法累积结果：

```
1: function SUM(L)
2:     s ← 0
3:     while L ≠ NIL do
4:         s ← s+ FIRST(L)
5:         L ← REST(L)
6:     return s

7: function PRODUCT(L)
8:     p ← 1
9:     while L ≠ NIL do
10:        p ← p · FIRST(L)
11:        L ← REST(L)
12:    return p
```

利用求积算法,我们可以将递归的阶乘实现转换为递推的方式 $n! = product([1..n])$。

1.2.8 最大值和最小值

如果非空有限列表中的元素可进行比较，则存在最大、最小值。max/min 的计算结构相同：

- 若列表中只有一个元素 $[x_1]$，结果为 x_1；
- 否则，递归地在子列表中寻找最大、最小值，并和表头元素比较得到最终结果。

$$\begin{aligned} min([x]) &= x \\ min(x : xs) &= \begin{cases} x < min(xs): & x \\ \text{否则}: & min(xs) \end{cases} \end{aligned} \tag{1.29}$$

和

$$\begin{aligned} max([x]) &= x \\ max(x:xs) &= \begin{cases} x > max(xs): & x \\ \text{否则}: & max(xs) \end{cases} \end{aligned} \tag{1.30}$$

这两个实现都从右向左处理，我们可以将其变为尾递归。并且这样还具备了“在线”处理能力，即任何时候，累积结果都是已处理部分中的最大、最小值。以 min 为例：

$$\begin{aligned} min'(a,\ \varnothing) &= a \\ min'(a,\ x:xs) &= \begin{cases} x < a: & min'(x, xs) \\ \text{否则}: & min'(a, xs) \end{cases} \end{aligned} \tag{1.31}$$

与 $sum'/prod'$ 不同，我们不能向 min'/max' 传入一个常数作为起始值，除非使用 $\pm\infty$（柯里化形式）：

$$min = min'(\infty) \qquad max = max'(-\infty)$$

为了解决这一问题，考虑到最大、最小值仅对非空列表有定义，可以将表头元素传入作为累积起始值：

$$min(x:xs) = min'(x,\ xs) \qquad max(x:xs) = max'(x,\ xs) \tag{1.32}$$

尾递归的最大、最小值算法可以进一步转化为迭代实现。我们略过 Max 以 Min 为例：

```
1: function Min(L)
2:     m ← First(L)
3:     L ← Rest(L)
4:     while L ≠ NIL do
5:         if First(L) < m then
6:             m ← First(L)
7:         L ← Rest(L)
8:     return m
```

还有一种尾递归实现，可以复用表头元素作为累积器。递归时，由于列表中至少有两个元素，我们每次拿出前两个比较，丢弃一个，然后继续处理剩余的元素。以 min 为例：

$$\begin{aligned} min([x]) &= x \\ min(x_1 : x_2 : xs) &= \begin{cases} x_1 < x_2: & min(x_1 : xs) \\ \text{否则}: & min(x_2 : xs) \end{cases} \end{aligned} \tag{1.33}$$

max 的实现与此对称。

练习 1.9

1. 使用尾递归实现 $length$
2. 使用尾递归实现插入排序。
3. 使用 n 的二进制形式，实现幂 b^n 的快速计算，使得复杂度为 $O(\lg n)$

1.3 变换

从代数结构的角度看，有两种不同的变换：一种保持列表结构，仅仅改变元素；另一种改变列表结构，变换结果和原列表不再同构。特别地，我们称保持列表结构的变换为**映射**。

1.3.1 逐一映射

我们通过一些例子来认识映射。第一个例子将一列数字映射为代表它们的字符串，如把 [3, 1, 2, 4, 5] 转换为 [“three”, “one”, “two”, “four”, “five”]。

$$\begin{aligned} toStr(\varnothing) &= \varnothing \\ toStr(x : xs) &= str(x) : toStr(xs) \end{aligned} \tag{1.34}$$

第二个例子是关于单词统计的。考虑一个字典，包含若干单词，并以它们的首字母分组，例如：

```
[[a, an, another, ... ],
 [bat, bath, bool, bus, ...],
 ...,
 [zero, zoo, ...]]
```

接下来我们处理一段文章，例如《哈姆莱特》(《王子复仇记》)，统计各单词在其中的出现次数，例如：

```
[[(a, 1041), (an, 432), (another, 802), ... ],
 [(bat, 5), (bath, 34), (bool, 11), (bus, 0), ...],
 ...,
 [(zero 12), (zoo, 0), ...]]
```

现在我们要找出，对应每个首字母，哪个单词被使用的次数最多？输出结果是一个单词列表，表中每个单词都是在各自首字母组中出现最多的一个，形如：`[a, but, can, ...]`。我们需要设计一个程序，将一组单词/次数对的列表转换成一个单词列表。

我们首先定义一个函数，接受一个单词/次数对列表，并找出现次数最多的单词。我们无需排序，只需要实现某种特殊映射的 $maxBy(cmp, L)$，其中 cmp 是抽象的比较函数。

$$\begin{aligned} maxBy(cmp,\ [x]) &= x \\ maxBy(cmp,\ x_1 : x_2 : xs) &= \begin{cases} cmp(x_1, x_2): & maxBy(cmp, x_2 : xs) \\ \text{否则}: & maxBy(cmp, x_1 : xs) \end{cases} \end{aligned} \tag{1.35}$$

对于一对值 $p = (a, b)$，我们定义如下辅助函数：

$$\begin{cases} fst\ (a, b) = & a \\ snd\ (a, b) = & b \end{cases} \tag{1.36}$$

为避免过多的括弧 $fst((a, b)) = a$，我们使用了空格。在上下文清楚的情况下，我们等价使用这两种记法：$f\ x = f(x)$。接下来就可以定义单词/次数对的比较函数了：

$$less(p_1, p_2) = snd(p_1) < snd(p_2) \tag{1.37}$$

将 $less$ 传入 $maxBy$ 就可选出出现次数最多的单词（柯里化形式）：

$$max'' = maxBy(less) \tag{1.38}$$

最后，我们调用 $max''()$ 处理单词统计列表：

$$\begin{aligned} solve(\varnothing) &= \varnothing \\ solve(x : xs) &= fst(max''(x)) : solve(xs) \end{aligned} \tag{1.39}$$

映射

尽管解决的问题不同，$solve()$ 和 $toStr()$ 反映出同样的计算结构。我们将这样的结构抽象为**映射**。

$$\begin{aligned} map(f,\ \varnothing) &= \varnothing \\ map(f,\ x : xs) &= f(x) : map(f, xs) \end{aligned} \tag{1.40}$$

map 接受一个函数 f 作为参数，然后将其应用到列表中的每个元素上。我们称将其它函数作为计算对象的函数为“高阶函数”。如果 f 的类型为 $A \to B$，即把类型 A 的元素映射为类型 B 的元素，则 map 的类型为：

$$map :: (A \to B) \to [A] \to [B] \tag{1.41}$$

读作：map 接受一个类型为 $A \to B$ 的函数，然后将一个类型为 $[A]$ 的列表变换为另一个类型为 $[B]$ 的列表。上面的两个例子可以使用映射定义如下（柯里化形式）：

$$\begin{aligned} toStr &= map\ str \\ solve &= map\ (fst \circ max'') \end{aligned}$$

其中 $f \circ g$ 表示函数组合，它首先应用函数 g，然后再应用函数 f，即 $(f \circ g)\ x = f(g(x))$。读作 f 作用于 g 之后。我们也可以从集合论的角度来定义映射。函数 $y = f(x)$ 定义了一个从集合 X 中的元素 x 到集合 Y 中的元素 y 的映射：

$$Y = \{f(x)|x \in X\} \tag{1.42}$$

这种形式的定义出的集合称为“策梅罗——弗兰克尔”集合抽象（简称 ZF 表达式）[72]。不同之处在于，我们定义的是从一个列表到另一个列表的映射 $Y = [f(x)|x \in Y]$，而不是集合的映射。其中可以含有重复的元素。列表的 ZF 表达式被称作“列表解析”。

列表解析是一个强大的工具。作为例子，我们思考如何实现一个排列算法。我们从全排列（[72]、[94]）出发，定义更一般的排列函数 $perm(L, r)$，列举从长度为 n 的列表 L 中选出 r 个元素的全部排列。一共有 $P_n^r = \dfrac{n!}{(n-r)!}$ 种不同的排列。

$$perm(L, r) = \begin{cases} |L| < r \text{ or } r = 0: & [[\ ]] \\ \text{否则}: & [x : ys \mid x \in L, ys \in perm(delete(x, L), r-1)] \end{cases} \tag{1.43}$$

如果选出 0 个元素排列，或者列表中元素的个数小于 r，排列结果为空列表的列表 [[]]；否则，我们逐一取出列表中每个元素 x，递归地从剩余的 $n-1$ 个元素中选择 $r-1$ 个元素排列，然后再将 x 置于每个排列的前面。下面的 Haskell 例子程序，使用了 ZF 表达式实现排列算法：

```
perm xs r | r == 0 || length xs < r = [[]]
          | otherwise = [ x:ys | x ←xs,
                                 ys ← perm (delete x xs) (r-1)]
```

为了简化映射的迭代实现，下面的算法中使用了一个辅助节点。

```
1: function MAP(f, L)
2:     L' ← CONS(⊥, NIL)                          ▷ 辅助节点
3:     p ← L'
4:     while L ≠ NIL do
5:         x ← FIRST(L)
6:         L ← REST(L)
7:         REST(p) ← CONS(f(x), NIL)
8:         p ← REST(p)
9:     return REST(L')                            ▷ 丢弃辅助节点
```

逐一执行

某些情况下我们只希望逐一处理表中的每个元素，而无需构造新的列表。例如打印一个列表中的每个元素：

```
1: function PRINT(L)
2:     while L ≠ NIL do
3:         print FIRST(L)
4:         L ← REST(L)
```

通常，我们在遍历时传入一个过程 P，然后遍历列表，将 P 应用到每个元素上。

```
1: function FOR-EACH(P, L)
2:     while L ≠ NIL do
3:         P(FIRST(L))
4:         L ← REST(L)
```

映射的例子

作为例子，我们思考一下“n 盏灯”的趣题 [96]：屋子里有 n 盏灯，都是熄灭的。我们执行下面的过程 n 次。

1. 将所有的灯都打开；
2. 扳动第 2、4、6……盏灯的开关。如果灯是亮的，则熄灭；如果是灭的，则点亮；
3. 每三个灯，扳动一次开关。第 3、6、9……位置上的灯的明暗状态切换；
4. ……

最后一轮的时候，只有最后一盏灯（第 n 盏）的开关被扳动。问最终有几盏灯是亮的?

先考虑最简单的穷举法。把 n 盏灯表示为一列 0、1 数字，其中 0 表示熄灭，1 表示点亮。开始时，灯都是灭的 [0, 0, ..., 0]。将灯编号为 1 到 n，然后映射成一个关于 (i, 亮/灭) 的列表：

$$lights = map(i \mapsto (i, 0),\ [1, 2, 3, ...n])$$

这一映射将每个编号都对应到 0 上，结果为一个列表，每个元素是一对值：$L = [(1, 0), (2, 0), ..., (n, 0)]$。然后我们操作这一列表 n 次。在第 i 次操作中，逐一检查每对值，如果灯的编号能被 i 整除，我们就将状态翻转。考虑 $1 - 0 = 1$ 且 $1 - 1 = 0$，我们可以将亮灭状态 x 的切换实现为 $1 - x$。对于灯 (j, x)，若 $i|j$（即 $j \bmod i = 0$），就翻转灯的状态，否则就跳过不做处理。

$$switch(i, (j, x)) = \begin{cases} j \bmod i = 0: & (j, 1 - x) \\ \text{否则}: & (j, x) \end{cases} \tag{1.44}$$

对所有灯的第 i 轮操作用映射实现为：

$$map(switch(i), L) \tag{1.45}$$

这里，我们使用了 $switch$ 的柯里化形式，它等价于：

$$map((j, x) \mapsto switch(i, (j, x)), L)$$

接下来，我们定义一个函数 $op()$，重复执行上述对 L 的映射 n 次。调用形式为：$op([1, 2, ..., n], L)$。

$$\begin{aligned} op(\varnothing,\ L) &= L \\ op(i : is,\ L) &= op(is, map(switch(i), L)) \end{aligned} \tag{1.46}$$

最后，将每对值的第二个元素累加起来就得到最终的答案。

$$solve(n) = sum(map(snd, op([1, 2, ..., n], lights))) \tag{1.47}$$

下面的 Haskell 例子程序实现了这一穷举解法。

```
solve = sum ∘ (map snd) ∘ proc  where
    lights = map (λi → (i, 0)) [1..n]
    proc n = operate [1..n] lights
    operate [] xs = xs
    operate (i:is) xs = operate is (map (switch i) xs)

switch i (j, x) = if j `mod` i == 0 then (j, 1 - x) else (j, x)
```

我们列出灯的数目为 1、2、……、100 盏时的答案（人为添加了换行）：

```
[1,1,1,
 2,2,2,2,2,
 3,3,3,3,3,3,3,
 4,4,4,4,4,4,4,4,4,
 5,5,5,5,5,5,5,5,5,5,5,
 6,6,6,6,6,6,6,6,6,6,6,6,6,
 7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,
 8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,
 9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,10]
```

这一结果很有规律：

- 3 盏灯以内时，最后仍然亮的灯为 1 盏；
- 4 盏灯到 8 盏灯时，最后仍然有 2 盏灯是亮的；
- 9 盏灯到 15 盏灯时，最后有 3 盏灯是亮的；
- ……

看起来，当灯的数目为 i^2 到 $(i+1)^2 - 1$ 盏时，最后会有 i 盏灯是亮的。事实上，我们可以证明这一结论：

证明. 将 n 盏灯编号为 1 到 n，考虑最后仍然亮的那些灯。由于初始时，所有灯都是灭的，我们可以确定，被扳动奇数次开关的灯最后是亮的。对于编号为 i 的灯，若 i 可以被 j 整除（表示为 $j|i$），则在第 j 轮，它的开关被扳动一次。所以当灯的编号含有奇数个因子时，最后的状态是亮的。

为了找出最后亮的灯，我们需要找出所有含有奇数个因子的数。对于任意自然数 n，记 S 为 n 的所有因子的集合。S 初始化为 $\varnothing$，若 p 为 n 的一个因子，则必然存在一个正整数 q，使得 $n = pq$。也就是说 q 也是 n 的因子。因此当且仅当 $p \neq q$ 时，我们向集合 S 中添加两个不同的因子，这样 $|S|$ 将总是偶数。除非 $p = q$，此时 n 必定是完全平方数。这时只能向集合 S 中增加一个因子，从而导致奇数个因子。 □

根据这一结论，我们可以通过寻找 n 以内的完全平方数来快速解决这一趣题。

$$solve(n) = \lfloor \sqrt{n} \rfloor \tag{1.48}$$

下面的 Haskell 例子程序输出 1、2、……、100 盏灯时的结果：

```
map (floor ∘ sqrt) [1..100]
```

映射是一个抽象概念，它不仅局限于列表，也可以扩展到许多复杂的代数结构。下一章我们会介绍如何对树结构进行映射。只要我们能够遍历一个结构，并且有空结构的定义，就可以使用映射的概念。

1.3.2 反转

如何用最小的空间反转一个单向链表是一道经典题目，需要仔细操作节点的引用。其实存在一个简单的策略：

1. 先写出一个纯递归解；
2. 转换为尾递归；
3. 将尾递归转换为命令式操作。

纯递归解很直观，为了反转列表 L。

- 若 L 为空，反转结果也为空；
- 否则，递归地反转子列表 L'，然后将第一个元素添加到尾部。

$$\begin{aligned} reverse(\varnothing) &= \varnothing \\ reverse(x : xs) &= append(reverse(xs), x) \end{aligned} \tag{1.49}$$

但是这一方法的性能不佳。每次需要遍历列表以在尾部添加元素。总体时间复杂度是平方级的。可以将其优化为尾递归形式。我们使用一个累积器来记录中间的反转结果，并传入空列表来启动反转 $reverse = reverse'(\varnothing)$。

$$\begin{aligned} reverse'(A,\ \varnothing) &= A \\ reverse'(A,\ x : xs) &= reverse'(x : A,\ xs) \end{aligned} \tag{1.50}$$

不同于在尾部添加，cons 是常数时间操作。我们不断从列表的头部逐一取出元素，将其置于累积结果的前面。这相当于将全部元素压入一个堆栈，然后再依次弹出。整体上是线性时间的。由于尾递归无需记录上下文环境，我们可以将其优化为循环迭代：

```
1: function REVERSE(L)
2:     A ← NIL
3:     while L ≠ NIL do
4:         A ← CONS(FIRST(L), A)
5:         L ← REST(L)
6:     return A
```

但是，这一算法生成了一个新的反转列表，而不是在原列表上直接修改。我们接下来要通过重用 L 将其改为就地修改的形式。如下面的例子程序：

```
List<T> reverse(List<T> xs) {
  List<T> p, ys = null
  while (xs ≠ null) {
    p = xs
    xs = xs.next
    p.next = ys
    ys = p
  }
  return ys
}
```

练习 1.10

1. 给定一个 0 到 10 亿以内的数，编程将其转换为英文表示，例如 123 转换为“one hundred and twenty three”，如果带有小数部分呢？
2. 使用尾递归求 $[(k, v)]$ 列表中 v 值最大的元素。

1.4 子列表

数组可以快速地分割为连续的切片，而列表分割则需要遍历，因而这类操作都是线性时间的。

1.4.1 截取、丢弃、分割

从列表中取出前 n 个元素,相当于获取从第 1 到第 n 个元素的的子列表：$sublist(1, n, L)$。如果 n 为 0 或 $L = \varnothing$，则子列表为空；否则，递归地在 L' 中取出 $n-1$ 个元素，再

将原头部元素置于最前。

$$\begin{aligned} take(0, L) &= \varnothing \\ take(n, \varnothing) &= \varnothing \\ take(n, x : xs) &= x : take(n - 1, xs) \end{aligned} \tag{1.51}$$

这一算法是这样处理越界情况的：如果 $n > |L|$ 或 n 为负数，最终转化为 L 为空的边界情况，返回整个列表作为结果。

从列表中丢弃前 n 个元素，等价于从右侧获取子列表 $sublist(n + 1, |L|, L)$，其中 $|L|$ 是列表的长度。它的实现和 $take$ 是对称的：

$$\begin{aligned} drop(0, L) &= L \\ drop(n, \varnothing) &= \varnothing \\ drop(n, x : xs) &= drop(n - 1, xs) \end{aligned} \tag{1.52}$$

我们把对应的迭代实现留作本节的练习。使用取出和丢弃操作，可以在列表的任何位置获取指定长度的子列表。

$$sublist(from, cnt, L) = take(cnt, drop(from - 1, L)) \tag{1.53}$$

另外一种形式，是传入左侧和右侧的边界：

$$slice(from, to, L) = drop(from - 1, take(to, L)) \tag{1.54}$$

边界的定义为闭区间 $[from, to]$，包括两端。我们也可以在指定位置把列表分割开：

$$splitAt(i, L) = (take(i, L), drop(i, L)) \tag{1.55}$$

练习 1.11

1. 将 $sublist$ 和 $slice$ 改写为柯里化形式，从而无需 L 作为参数。

条件截取和丢弃

take 与 drop 指定截取或丢弃的个数。我们可以对其扩展，只要某种条件成立，就不断取出或者丢弃元素，称为 $takeWhile/dropWhile$。它们逐一检查元素是否满足给定条件，如果不满足，则停止检查剩余的部分，这和后面介绍的过滤算法有所不同。

$$\begin{aligned} takeWhile(p,\ \varnothing) &= \varnothing \\ takeWhile(p,\ x : xs) &= \begin{cases} p(x): & x : takeWhile(p, xs) \\ \text{否则}: & \varnothing \end{cases} \end{aligned} \tag{1.56}$$

其中 p 是判断条件。将 p 应用到一个元素上，结果是真或假表示条件是否满足。$dropWhile$ 的实现是对称的：

$$\begin{aligned} dropWhile(p,\ \varnothing) &= \varnothing \\ dropWhile(p,\ x:xs) &= \begin{cases} p(x): & dropWhile(p, xs) \\ \text{否则}: & x:xs \end{cases} \end{aligned} \tag{1.57}$$

1.4.2 切分和分组

切分和分组操作将列表中的元素重新安排成若干子列表。通常一边遍历一边进行这种分类安排，使得时间复杂度为线性。

切分

切分可以被认为是一种特殊的 split，我们不是在指定的位置将列表分开，而是检查每个元素是否满足某一条件，根据条件找出最长前缀。切分结果是一对子列表，一个是最长前缀，另一个包含剩余的部分。

切分有两种类型：一种是满足条件的最长子列表；另一种是**不**满足条件的最长子列表。前者称为 $span$，后者称为 $break$。

$$\begin{aligned} span(p,\ \varnothing) &= (\varnothing, \varnothing) \\ span(p,\ x:xs) &= \begin{cases} p(x): & (x:A,\ B) \text{ 其中 } (A, B) = span(p, xs) \\ \text{否则}: & (\varnothing,\ x:xs) \end{cases} \end{aligned} \tag{1.58}$$

只需要把条件取逻辑非，就可以实现 $break$：

$$break(p) = span(\neg p) \tag{1.59}$$

$span$ 和 $break$ 都寻找最长**前缀**。一旦条件打破就立即停下，忽略剩余部分。下面是 Span 的迭代实现：

```
1: function SPAN(p, L)
2:     A ← NIL
3:     while L ≠ NIL and p(FIRST(L)) do
4:         A ← CONS(FIRST(L), A)
5:         L ← REST(L)
6:     return (A, L)
```

这一算法创建了一个新的列表用以存放最长前缀，我们也可以复用原列表的空间，将其转换为就地修改的实现。

```
1: function SPAN(p, L)
2:     A ← L
```

```
3:    tail ← NIL
4:    while L ≠ NIL and p(First(L)) do
5:        tail ← L
6:        L ← Rest(L)
7:    if tail = NIL then
8:        return (NIL, L)
9:    Rest(tail) ← NIL
10:   return (A, L)
```

分组

span 和 *break* 将列表切分为两部分，分组将列表中的元素分成若干子列表。例如将一个字符串分割为若干单位，每个包含连续相同的字符：

```
group ``Mississippi'' = [``M'', ``i'', ``ss'', ``i'',
                         ``ss'',``i'', ``pp'', ``i'']
```

再例如，给出一列数字：

$$L = [15, 9, 0, 12, 11, 7, 10, 5, 6, 13, 1, 4, 8, 3, 14, 2]$$

把它分成若干组，每组中的元素都按降序排列：

$$group(L) = [[15, 9, 0], [12, 11, 7], [10, 5], [6], [13, 1], [4], [8, 3], [14, 2]]$$

这两个例子都具有实用价值。字符串分组后，可用于构造基数树这种数据结构，用于快速的文本搜索。有序子列表可用于实现自然归并排序。我们将在后继章节介绍它们。

我们把分组条件抽象成某种关系 $\sim$。它用于判断两个相邻元素 x、y 是否“等价”：$x \sim y$。我们遍历列表，每次比较两个元素。如果等价，就把它们置于一组；否则仅把 x 置于组内，而把 y 置于一个新组中。

$$\begin{aligned} group(\sim, \varnothing) &= [\varnothing] \\ group(\sim, [x]) &= [[x]] \\ group(\sim, x : y : xs) &= \begin{cases} x \sim y : & (x : ys) : yss \\ \text{否则} : & [x] : ys : yss \end{cases} \end{aligned} \tag{1.60}$$

其中 $(ys : yss) = group(\sim, xs)$。这一算法的时间复杂度为 $O(n)$，其中 n 是长度。也可以用迭代的方式实现这一算法。若 L 不为空，我们将分组结果初始化为 $[[x_1]]$，其中 x_1 是表头元素。然后从第二个元素开始遍历列表，若相邻的两个元素“等价”，我们就将遍历到的元素放入最后一组，否则就新创建一个组。

```
1: function Group(∼, L)
```

```
    if L = NIL then
        return [NIL]
    x ← First(L)
    L ← Rest(L)
    g ← [x]
    G ← [g]
    while L ≠ NIL do
        y ← First(L)
        if x ∼ y then
            g ← Append(g, y)
        else
            g ← [y]
            G ← Append(G, g)
        x ← y
        L ← Next(L)
    return G
```

如果添加操作 append 没有尾部引用优化，这一实现的时间复杂度会退化为平方级别。如果不关心顺序，可以将 append 改为 cons。使用分组函数，本节开头的两个例子就可以实现如下：

$$group(=, [m, i, s, s, i, s, s, i, p, p, i]) = [[M], [i], [ss], [i], [ss], [i], [pp], [i]]$$

和

$$\begin{aligned} & group(\geq, [15, 9, 0, 12, 11, 7, 10, 5, 6, 13, 1, 4, 8, 3, 14, 2]) \\ & = [[15, 9, 0], [12, 11, 7], [10, 5], [6], [13, 1], [4], [8, 3], [14, 2]] \end{aligned}$$

也可以使用 *span* 来实现分组。传入一个条件，*span* 将列表分割成两部分：其中之一是满足条件的最长子列表。我们对剩余部分不断执行 span，直到处理完所有元素。但是传入 span 的条件函数是一元函数，它只接受一个元素作为参数，检查它是否满足。而分组条件要求是二元函数。它接受两个元素并进行比较。可以用柯里化来解决这一差异：将第一个元素传入二元判断函数并固定，然后用柯里化后的函数判断剩余的元素。

$$\begin{aligned} group(\sim, \varnothing) &= [\varnothing] \\ group(\sim, x : xs) &= (x : A) : group(\sim, B) \end{aligned} \tag{1.61}$$

其中 $(A, B) = span(y \mapsto x \sim y, xs)$，是对子列表进行 span 的结果。虽然这个新分组函数可以将单词中的相同字母分组：

```
group (==) ``Mississippi''
[``m'', ``i'', ``ss'', ``i'', ``ss'', ``i'', ``pp'', ``i'']
```

但它却不能正确地将数字按照降序分组：

```
group (≥) [15, 9, 0, 12, 11, 7, 10, 5, 6, 13, 1, 4, 8, 3, 14, 2]
[[15,9,0,12,11,7,10,5,6,13,1,4,8,3,14,2]]
```

第一个元素是 15，它被置于 $\geq$ 的左侧进行比较。15 是列表中的最大元素，因此 span 把所有元素都置于 A 中，而 B 为空。这并不是错误，而是正确的行为。因为分组被设计为将“等价”的元素放到一起。严格说来，等价关系（$\sim$）条件必须满足三个性质：自反性、传递性、对称性。

1. **自反性**：$x \sim x$，即任何元素和它自己等价；
2. **传递性**：$x \sim y, y \sim z \Rightarrow x \sim z$，如果两个元素等价，并且其中之一和第三个元素等价，则这三个元素等价；
3. **对称性**：$x \sim y \Leftrightarrow y \sim x$，即比较的顺序不影响结果。

对“Mississippi”分组时，我们使用等号（$=$）作为判断条件，上述三个条件都被满足。这自然产生了正确的结果。但将柯里化的大于等于号（$\geq$）作为等价条件传入时，就违反了自反性和对称性。因而无法按照预期对数字进行分组。用 span 实现的第二个分组算法，将含义限制为更严格的等价关系，而第一个分组算法则无此种限制。它仅检查任何两个相邻元素是否满足条件，这比等价条件要弱。

练习 1.12

1. 修改 $take/drop$ 算法，当 n 是负数时，$take$ 返回 $\varnothing$，$drop$ 返回全部列表。
2. 实现就地修改的 take 和 drop 算法。
3. 实现 takeWhile 和 dropWhile 算法。
4. 考虑下面 $span$ 的实现：

$$
\begin{aligned}
span(p,\ \varnothing) &= (\varnothing, \varnothing) \\
span(p,\ x:xs) &= \begin{cases} p(x): & (x:A, B) \\ \text{否则}: & (A, x:B) \end{cases}
\end{aligned}
$$

其中 $(A, B) = span(p, xs)$，它和我们本节给出的实现有何不同？

1.5 叠加

几乎所有的列表算法都有着共同的结构。这不是一个巧合。这种共性本质上来自列表的递归性质。我们可以将列表算法抽象到更高层次的概念：叠加[5]。它本质上是所有列表计算的初始代数 [99]。

[5] 也叫作 reduce

1.5.1 右侧叠加

比较 sum、$product$、$sort$，它们都有共同的结构。

$$\begin{aligned} h(\varnothing) &= z \\ h(x:xs) &= x \oplus h(xs) \end{aligned} \tag{1.62}$$

我们可以将两个部分抽象出来：

- 列表为空时的结果。求和时为 0；求积时为 1；排序时为 $\varnothing$；
- 对表头元素和递归结果进行计算的二元操作。求和时是相加；求积时是相乘；排序时是按序插入。

我们将空列表时的结果抽象为**初始值**，记为 z（代表抽象的零），二元运算抽象为 $\oplus$。上述定义可以参数化为：

$$\begin{aligned} h(\oplus, z,\ \varnothing) &= z \\ h(\oplus, z,\ x:xs) &= x \oplus h(\oplus, z, xs) \end{aligned} \tag{1.63}$$

我们输入列表 $L = [x_1, x_2, ..., x_n]$，将计算过程展开如下：

$$\begin{aligned} & h(\oplus, z, [x_1, x_2, ..., x_n]) \\ = \ & x_1 \oplus h(\oplus, z, [x_2, x_3, ..., x_n]) \\ = \ & x_1 \oplus (x_2 \oplus h(\oplus, z, [x_3, ..., x_n])) \\ & ... \\ = \ & x_1 \oplus (x_2 \oplus (...(x_n \oplus h(\oplus, z, \varnothing))...)) \\ = \ & x_1 \oplus (x_2 \oplus (...(x_n \oplus z)...)) \end{aligned}$$

这些括弧是必须的，它限制计算顺序从最右侧开始（$x_n \oplus z$），不断向左侧进行直到 x_1。这和图1.3描述的折扇相似。折扇由竹子和纸制成。多根扇骨在末端被轴穿在一起。把展开的扇形逐渐折叠，最终收起成一根。

图 1.3: 折扇

这些扇骨组成了一个列表。收起扇子的二元操作是将一根扇骨叠在已收起的部分之上。最初的收起部分为空。收起的过程从一端开始，不断应用二元操作，直到所有的扇骨都叠在一起。求和与求积算法和收起折扇的过程是相同的。

$$\begin{aligned} sum([1,2,3,4,5]) &= 1+(2+(3+(4+5))) \\ &= 1+(2+(3+9)) \\ &= 1+(2+12) \\ &= 1+14 \\ &= 15 \end{aligned}$$

$$\begin{aligned} product([1,2,3,4,5]) &= 1\times(2\times(3\times(4\times5))) \\ &= 1\times(2\times(3\times20)) \\ &= 1\times(2\times60) \\ &= 1\times120 \\ &= 120 \end{aligned}$$

我们称这一过程为**叠加**。特别地，由于计算从右侧一端开始，我们将其记为 $foldr$：

$$\begin{aligned} foldr(f,z,\ \varnothing) &= z \\ foldr(f,z,\ x:xs) &= f(x,foldr(f,z,xs)) \end{aligned} \tag{1.64}$$

使用 $foldr$，求和与求积可以定义如下：

$$\begin{aligned} \sum_{i=1}^{n} x_i &= x_1+(x_2+(x_3+...+(x_{n-1}+x_n))...) \\ &= foldr(+,0,[x_1,x_2,...,x_n]) \end{aligned} \tag{1.65}$$

$$\begin{aligned} \prod_{i=1}^{n} x_i &= x_1\times(x_2\times(x_3\times...+(x_{n-1}\times x_n))...) \\ &= foldr(\times,1,[x_1,x_2,...,x_n]) \end{aligned} \tag{1.66}$$

或者写成柯里化形式：$sum = foldr(+,0)$ 和 $product = foldr(\times,1)$。插入排序算法也可用 $foldr$ 定义为：

$$sort = foldr(insert,\varnothing) \tag{1.67}$$

1.5.2 左侧叠加

我们可以把 $foldr$ 转换为尾递归。它产生同样的结果，但是计算是从左向右进行的。因此我们将其记为 $foldl$：

$$\begin{aligned} foldl(f,z,\ \varnothing) &= z \\ foldl(f,z,\ x:xs) &= foldl(f,f(z,x),xs) \end{aligned} \tag{1.68}$$

以 sum 为例，可以看到计算是从何自左向右展开的：

$$
\begin{aligned}
& foldl(+, 0, [1, 2, 3, 4, 5]) \\
= \quad & foldl(+, 0 + 1, [2, 3, 4, 5]) \\
= \quad & foldl(+, (0 + 1) + 2, [3, 4, 5]) \\
= \quad & foldl(+, ((0 + 1) + 2) + 3, [4, 5]) \\
= \quad & foldl(+, (((0 + 1) + 2) + 3) + 4, [5]) \\
= \quad & foldl(+, ((((0 + 1) + 2 + 3) + 4 + 5, \varnothing) \\
= \quad & 0 + 1 + 2 + 3 + 4 + 5
\end{aligned}
$$

每一步都推迟了 $f(z, x)$ 的计算，这是惰性求值时的行为。否则每次调用的求值序列为 $[1, 3, 6, 10, 15]$。一般来说，$foldl$ 可以展开为下面的形式：

$$foldl(f, z, [x_1, x_2, ..., x_n]) = f(f(...(f(f(z, x_1), x_2), ..., x_n) \tag{1.69}$$

或采用中缀记法：

$$foldl(\oplus, z, [x_1, x_2, ..., x_n]) = z \oplus x_1 \oplus x_2 \oplus ... \oplus x_n \tag{1.70}$$

$foldl$ 是尾递归的，可以将其实现为循环。我们将结果初始化为 z，然后把二元运算应用于结果和每个元素上。命令式算法通常称作 REDUCE。

```
1: function REDUCE(f, z, L)
2:     while L ≠ NIL do
3:         z ← f(z, FIRST(L) )
4:         L ← REST(L)
5:     return z
```

$foldr$ 和 $foldl$ 各自有适合的应用场景，它们并不总能互换。例如，某些容器只支持从一端添加元素（如栈）。我们要定义一个 $fromList$ 函数，从一个列表构建出容器（柯里化形式）：

$$fromList = foldr(add, empty)$$

其中 $empty$ 是空容器。单向链表本身就是这样一种容器。在头部添加元素的性能要远高于在尾部添加。如果希望复制列表并保持顺序，$foldr$ 就是一个自然的选择，而 $foldl$ 则产生逆序的列表。在迭代实现时为了解决逆序问题，我们可以先反转列表，再执行 reduce 操作：

```
1: function REDUCE-RIGHT(f, z, L)
2:     return REDUCE(f, z, REVERSE(L))
```

有人认为应优先使用 $foldl$，因为它是尾递归的，同时满足函数式和命令式场景，并且还是在线算法。但在处理无穷列表（用流和惰性求值实现）时，就只能用 $foldr$。例如，下面的例子程序将一个无穷列表中的每个元素都置于一个单独的列表中，并返回前 10 个：

$$take(10, foldr((x, xs) \mapsto [x] : xs, \varnothing, [1, 2, ...])$$
$$\Rightarrow [[1], [2], [3], [4], [5], [6], [7], [8], [9], [10]]$$

这里不能使用 $foldl$，因为外层计算永远不会完成。当左右没有区别时，我们用统一的符号 $fold$。本书也使用符号 $fold_l$ 和 $fold_r$ 来强调叠加本身而非方向。尽管本章内容是关于列表的，但是叠加概念是抽象的。它可以应用到其它代数结构。我们可以对一棵树（[99]2.6 节）、一个队列、和更复杂的结构进行叠加。只要它满足下面这两个条件：

- 定义了空（例如空树）；
- 可分解的递归结构（例如一棵树可分解为子树和元素）。

人们进一步将这些概念抽象为可叠加、幺半群、可遍历等等。

练习 1.13

1. 为了用 $foldr$ 定义插入排序，我们将插入函数设计成 $insert(x, L)$，这样排序可以表示为：$sort = foldr(insert, \varnothing)$。$foldr$ 的类型为：

$$foldr :: (A \to B \to B) \to B \to [A] \to B$$

其中第一个参数 f 的类型是：$A \to B \to B$，初始值 z 的类型为 B。它对元素类型为 A 的列表进行叠加，最终结果的类型为 B。如何用 $foldl$ 定义插入排序？$foldl$ 的类型是什么？

1.5.3 例子

作为例子，我们来看如何用 $fold$ 和 map 来解决 n 盏灯趣题。在穷举法中，我们创建了一个列表，每个元素是一对值 (i, s)，包含灯的序号 i 和明暗状态 s。每轮中操作中，如果灯的序号 i 能被轮数 j 整除，就翻转灯的状态。这一过程可以用 $fold$ 定义：

$$fold_r(step, [(1, 0), (2, 0), ..., (n, 0)], [1, 2, ..., n])$$

初始时所有灯都是灭的。要折叠的列表是从 1 到 n 的轮数。函数 $step$ 接受两个参数：一个是轮数 i，另一个是“灯序号/状态”对列表。我们用 map 来翻转灯的状态：

$$fold_r((i, L) \mapsto map(switch(i), L), [(1, 0), (2, 0), ..., (n, 0)], [1, 2, ..., n])$$

$fold_r$ 的结果是最终的序号/明暗状态对列表。接下来用 map 从每对值中提取出状态，再用 sum 求出有几盏灯是点亮的：

$$sum(map(snd, \quad fold_r((i, L) \mapsto map(switch(i), L), [(1, 0), (2, 0), ..., (n, 0)], [1, 2, ..., n]))) \tag{1.71}$$

串联

如果让 $fold$ 用“$+\!\!+$”（第1.2.6节）作用在一组列表上，其结果相当于把它们串联成一个列表。这和数字累加是类似的过程：

$$concat = fold_r(+\!\!+, \varnothing) \tag{1.72}$$

这是柯里化的定义，其用法如下：

$$concat([[1], [2, 3, 4], [5, 6, 7, 8, 9]]) \Rightarrow [1, 2, 3, 4, 5, 6, 7, 8, 9]$$

练习 1.14

1. $concat$ 的时间复杂度是什么？
2. 设计一个线性时间的 $concat$ 算法。
3. 使用 $foldr$ 来定义 map。

1.6 搜索和过滤

搜索和过滤都是抽象概念，不仅限于列表，它们也可用于更广泛的对象。对于列表，它们通常需要遍历以找到结果。

1.6.1 属于

给定类型 A 的元素 a，如何检查它是否属于某个元素类型为 A 的列表？我们可以遍历列表将每个元素和 a 进行对比，直到发现相等的元素或到达尾部。

- 若列表为空，则 a 不存在；
- 若表头元素等于 a，则存在；
- 否则，递归地检查 a 是否属于子列表。

$$\begin{aligned} a \in \varnothing &= False \\ a \in (b : bs) &= \begin{cases} b = a : & True \\ b \neq a : & a \in bs \end{cases} \end{aligned} \tag{1.73}$$

这一算法也称作 $elem$。它的复杂度为 $O(n)$，其中 n 是长度。若列表有序（例如升序），有的人会想用分而治之的方法将其优化到对数时间。但列表不支持常数时间的随机访问，无法使用二分查找（见第 3 章）。

1.6.2 查询

我们接下来扩展 $elem$ 操作。在 n 盏灯趣题中，我们使用了“键/值”对列表 $[(k, v)]$。每对元素包含一个键、一个值。这种列表称作“关联列表”。如果在其中查询某个值，我们需要把值的部分提取出来进行比较。

$$\begin{aligned} lookup(x, \varnothing) &= Nothing \\ lookup(x, (k, v) : kvs) &= \begin{cases} v = x : & Just\ (k, v) \\ v \neq x : & lookup(x, kvs) \end{cases} \end{aligned} \tag{1.74}$$

和 $elem$ 不同，我们不仅想知道存在与否，还希望返回找到的键/值对。由于待查询的值并不一定存在，我们引入了一种称作“可能”的代数类型，**Maybe** A 类型有两种不同的值：或是类型 A 的某个值 a，或是空。分别记为 $Just\ a$ 或 $Nothing$。这是一种解决空引用的方法（[99]4.2.2 节）。

1.6.3 查找和过滤

我们可以进一步扩展 $lookup$ 到一般情况。不再仅仅比较元素是否等于待查询的值，而是查找满足某一条件的元素：

$$\begin{aligned} find(p,\ \varnothing) &= Nothing \\ find(p,\ (x : xs)) &= \begin{cases} p(x) : & Just\ x \\ \text{否则} : & find(p, xs) \end{cases} \end{aligned} \tag{1.75}$$

尽管可能有多个元素满足条件，$find$ 只返回第一个。我们可以把它扩展为查找全部满足条件的元素，这一过程通常称作过滤，如图1.4所示。

图 1.4: 输入：$[x_1, x_2, ..., x_n]$，输出：$[x'_1, x'_2, ..., x'_m]$。满足：$\forall x'_i \Rightarrow p(x'_i)$.

我们也可以用 ZF 表达式来定义 $filter$：

$$filter(p, X) = [x_i | x_i \in X, p(x_i)] \tag{1.76}$$

和 $find$ 不同，如果没有任何元素满足条件，$filter$ 返回空列表。它逐一扫描列表，检查每个元素：

$$\begin{aligned} filter(p,\ \varnothing) &= \varnothing \\ filter(p,\ x : xs) &= \begin{cases} p(x) : & x : filter(p, xs) \\ \text{否则} : & filter(p, xs) \end{cases} \end{aligned} \tag{1.77}$$

这一算法从右向左构造结果。在迭代实现中，如果用 $append$ 来构造结果，性能会下降到 $O(n^2)$。

```
1: function FILTER(p, L)
2:     L' ← NIL
3:     while L ≠ NIL do
4:         if p(FIRST(L)) then
5:             L' ← APPEND(L', FIRST(L))                ▷ 线性时间
6:         L ← REST(L)
```

正确的做法是用 cons 替代，但这样返回的结果是逆序的。我们可以再执行一次线性时间的反转（见练习）。从右向左进行计算的性质提示我们可以用 $foldr$ 来定义过滤。我们需要设计一个函数 f 检查每个元素，如果符合条件就添加到结果中：

$$f(x, A) = \begin{cases} p(x): & x : A \\ \text{否则}: & A \end{cases} \tag{1.78}$$

我们需要将判定条件 p 传入 f。这样一共有 3 个参数 $f(p, x, A)$。将其柯里化就得到用 $foldr$ 定义的过滤算法：

$$filter(p) = foldr((x, A) \mapsto f(p, x, A), \varnothing) \tag{1.79}$$

我们可以进一步将其简化为（称作 η 变换 [73]）：

$$filter(p) = foldr(f(p), \varnothing) \tag{1.80}$$

过滤也是一个通用的概念。不仅限于列表，我们可以对任何可遍历的结构应用一个判定条件，获得感兴趣的信息。

1.6.4 匹配

匹配一般是指在一个结构中寻找某一模式。即使限定为列表和字符串，匹配仍是一个广泛、深入的内容。本书专门有章节介绍字符串匹配。这里我们仅仅考虑给定一个列表 A，检查它是否出现在另一个列表 B 中的情况。这里有两个特殊情况：判断 A 是否是 B 的前缀和后缀。式 (1.58) 介绍的 $span$ 算法寻找符合某个条件的最长前缀。我们可以使用类似的方法：逐一比较 A、B 中的每个元素直到遇到不同元素或到达任一列表尾部。若 A 是 B 的前缀，则记为：$A \subseteq B$。

$$\begin{aligned} \varnothing \subseteq B &= True \\ (a : as) \subseteq \varnothing &= False \\ (a : as) \subseteq (b : bs) &= \begin{cases} a \neq b: & False \\ a = b: & as \subseteq bs \end{cases} \end{aligned} \tag{1.81}$$

由于扫描列表，前缀检查是线性时间的。但是我们不能用同样的方法来检查后缀：对齐两个列表的尾部，然后从右向左倒序比较。这样的代价很大。这一点与数组

不同。为了实现线性时间的后缀检查，我们可以将两个列表都反转，然后使用前缀检查进行判断：

$$A \supseteq B = reverse(A) \subseteq reverse(B) \tag{1.82}$$

使用 $\subseteq$，就可以判断一个列表是否是另外一个的子列表。称作中缀检查。方法就是遍历目标列表，不断进行前缀检查：

$$\begin{aligned} infix?(a:as,\ \varnothing) &= False \\ infix?(A,\ B) &= \begin{cases} A \subseteq B: & True \\ \text{否则}: & infix?(A, B') \end{cases} \end{aligned} \tag{1.83}$$

若 A 为空，定义空列表是任何列表的中缀。由于 $\varnothing \subseteq B$ 总成立，因此算法给出正确结果。对于 $infix?(\varnothing, \varnothing)$，结果也是正确的。下面是对应的迭代实现：

```
1: function IS-INFIX(A, B)
2:     if A = NIL then
3:         return TRUE
4:     n ← |A|
5:     while B ≠ NIL and n ≤ |B| do
6:         if A ⊆ B then
7:             return TRUE
8:         B ← REST(B)
9:     return FALSE
```

由于前缀检测需要线性时间，并且在遍历时被不断调用，这一算法的复杂度为 $O(nm)$，其中 n 和 m 分别是两个列表的长度。即使替换成数组，如何将这种逐一比较的算法优化成线性时间仍是一个专门问题。第 13 章介绍了一些巧妙的方法，例如 KMP（Knuth-Morris-Pratt）算法，Boyer-Moore 算法。附录 C 介绍了后缀树方法。

对称地，我们可以枚举出 B 的所有后缀，然后检查 A 是否是某个后缀的前缀：

$$infix?(A, B) = \exists S \in suffixes(B), A \subseteq S \tag{1.84}$$

下面的 Haskell 例子程序使用列表解析实现了这一方法：

```
isInfixOf a b = (not ∘ null) [ s | s ← tails(b), a `isPrefixOf` s]
```

其中 `isPrefixOf` 进行前缀检查。`tails` 枚举一个列表的所有后缀。我们将其实现作为练习。

练习 1.15

1. 实现线性时间的属于（存在检查）算法。
2. 实现迭代的查询算法。

3. 使用 $reverse$ 实现线性时间的过滤算法。
4. 实现迭代的前缀检查算法。
5. 给定一个列表，枚举出它的所有后缀。

1.7 zip 和 unzip

关联列表常被作为一种字典的简易实现，用以处理少量数据。相对于树或者堆，其实现简单，但查询的性能是线性的而非对数的。在 n 盏灯趣题中，我们用下列方法创建关联列表：

$$map(i \mapsto (i, 0),\ [1, 2, ..., n])$$

我们经常需要将两个列表“关联”起来，为此可以定义一个 zip 函数：

$$\begin{aligned} zip(A,\ \varnothing) &= \varnothing \\ zip(\varnothing,\ B) &= \varnothing \\ zip(a : as,\ b : bs) &= (a, b) : zip(as, bs) \end{aligned} \tag{1.85}$$

这一算法可以处理长度不同的列表。关联结果的长度将和较短的一个相同。我们甚至可以关联无穷列表（如果两个列表都是无穷的，则需要惰性求值），例如[6]：

$$zip([0, 0, ...], [1, 2, ..., n])$$

给定单词列表，我们可以给每个单词顺序编号如下。

$$zip([1, 2, ...], [a, an, another, ...])$$

zip 从右向左构建结果。我们可以用 $foldr$ 定义它。算法的时间复杂度为 $O(m)$，其中 m 是较短列表的长度。在迭代实现时，如果使用 $append$ 操作，其性能会下降为平方时间，除非使用尾部引用优化。

```
1: function ZIP(A, B)
2:     C ← NIL
3:     while A ≠ NIL and B ≠ NIL do
4:         C ← APPEND(C, (FIRST(A), FIRST(B)))          ▷ 线性时间
5:         A ← REST(A)
6:         B ← REST(B)
7:     return C
```

为了避免 $append$，我们可以使用 cons，然后再把结果反转。但这样无法处理两个无穷列表。在命令式环境中，我们可以复用 A 来存储结果（视为一种映射：将一列元素映射为一列元素对）。

[6]在 Haskell 中：`zip (repeat 0) [1..n]`

我们可以进一步扩展 zip 关联多个列表。有些编程库提供了 `zip`、`zip3`、`zip4` ……直到 `zip7`。有些情况下，我们不是要构建元素对，而是要应用某种组合函数。例如，给定每种水果单价的列表，对于苹果、橙子、香蕉……其单价为 $[1.00, 0.80, 10.05, ...]$（单位是元）；顾客购买水果数量为：$[3, 1, 0, ...]$，表示购买了 3 个苹果，1 个橙子、0 个香蕉。下面的程序计算应付金额：

$$\begin{aligned} pays(U, \varnothing) &= \varnothing \\ pays(\varnothing, Q) &= \varnothing \\ pays(u : us, q : qs) &= (u \cdot q) : pays(us, qs) \end{aligned}$$

除了用乘法代替 cons，其计算结构和 zip 相同。我们将组合函数抽象为 f 并传给 zip，就定义出一个一般算法：

$$\begin{aligned} zipWith(f, A, \varnothing) &= \varnothing \\ zipWith(f, \varnothing, B) &= \varnothing \\ zipWith(f, a : as, b : bs) &= f(a, b) : zipWith(f, as, bs) \end{aligned} \tag{1.86}$$

下面是利用 $zipWith$ 定义内积（也称作点积）[98] 的例子：

$$A \cdot B = sum(zipWith(\cdot, A, B)) \tag{1.87}$$

zip 的逆运算是 $unzip$，将关联列表分解成两个列表。下面使用 $foldr$ 给出的柯里化定义：

$$unzip = foldr((a, b), (A, B) \mapsto (a : A, b : B), (\varnothing, \varnothing)) \tag{1.88}$$

我们从一对空列表开始叠加，不断将元素对分解为 a, b 并置于两个结果列表之前。分解过程也可以用 fst、snd 表达如下：

$$(p, P) \mapsto (fst(p) : fst(P), snd(p) : snd(P))$$

对于买水果的例子，若单价信息以关联列表的形式给出：

$$U = [(apple, 1.00), (orange, 0.80), (banana, 10.05), ...]$$

这样可用水果查到单价，如：$lookup(melon, U)$。购买数量也是关联列表：$Q = [(apple, 3), (orange, 1), (banana, 0), ...]$。计算总金额时，我们从两个关联列表中分解出单价和数量，然后计算其内积：

$$pay = sum(zipWith(\cdot, snd(unzip(U)), snd(unzip(Q)))) \tag{1.89}$$

$zipWith$ 结合惰性求值还可以定义无穷斐波那契数列：

$$F = 0 : 1 : zipWith(+, F, F') \tag{1.90}$$

它的含义是 F 是无穷的斐波那契数列，第一个元素是 0，第二个元素是 1。F' 是去掉头部元素的无穷斐波那契数列。从第三个元素起，每个斐波那契数，都是 F 和 F' 中对应元素的和。下面的例子程序列出了前 15 个斐波那契数：

```
fib = 0 : 1 : zipWith (+) fib (tail fib)

take 15 fib
[0,1,1,2,3,5,8,13,21,34,55,89,144,233,377]
```

zip 和 $unzip$ 的概念也是抽象的。我们可以扩展 zip 以关联两棵树，节点中的数据是成对元素，分别来自两棵树中。抽象的 zip 和 $unzip$ 还可以用于跟踪复杂结构的遍历路径，从而模拟命令式环境中的父节点引用（[10] 的最后一章）。

练习 1.16

1. 设计 iota 算法（I）其用法如下：
   - $iota(..., n) = [1, 2, 3, ..., n]$;
   - $iota(m, n) = [m, m + 1, m + 2, ..., n]$, where $m \leq n$;
   - $iota(m, m + a, ..., n) = [m, m + a, m + 2a, ..., n]$;
   - $iota(m, m, ...) = repeat(m) = [m, m, m, ...]$;
   - $iota(m, ...) = [m, m + 1, m + 2, ...]$.

   其中最后两个例子涉及无穷序列。可以通过流和惰性求值实现（[63] 和 [10]）。

2. 实现线性时间的命令式 zip 算法。
3. 用 $foldr$ 定义 zip。
4. 对于买水果的例子，如果购买数量的关联列表只包含非零的物品。不是

$$Q = [(apple, 3), (banana, 0), (orange, 1), ...]$$

   而是

$$Q = [(apple, 3), (orange, 1), ...]$$

   由于没有买香蕉，所以列表中没有香蕉相关的数据。编写程序计算总金额。

1.8 扩展阅读

列表是构建复杂数据结构和算法的基础，对于函数式编程尤为重要。我们介绍了构建、分解、更改、变换列表的基本算法；介绍了如何在列表中搜索、过滤、进行计算。尽管大多数编程环境都提供了预置的工具来支持列表，我们不应该仅仅把它们当作一些黑盒子。Rabhi 和 Lapalme 在 [72] 中介绍了关于列表的许多函数式算法。Haskell 标准库提供了关于基础算法的详细文档。伯德在 [1] 中给出了很多与叠加相关的例子，并介绍了“叠加融合定律”。

练习 1.17

1. 编写一个程序从列表中去除重复的元素。在命令式环境中，请用就地修改的方式删除这些重复元素。在纯函数环境中，构建一个只含有不同元素的新列表。结果列表中的元素顺序应保持和原列表中的一致。这一算法的复杂度是怎样的？如果允许使用额外的数据结构，可以如何简化实现？
2. 可以用列表来表示十进制的非负整数。例如 1024 可以表示为："$4 \to 2 \to 0 \to 1$"。一般来说，$n = d_m ... d_2 d_1$ 可以表示为："$d_1 \to d_2 \to ... \to d_m$"。任给两个用列表表示的数 a 和 b。实现它们的基本算数运算，例如加和减。
3. 在命令式环境中，循环列表是一种有缺陷的列表：某个节点指向了以前的位置，如图1.5所示。当遍历的时候，会陷入无限循环。设计一个算法检查某个列表是否含有循环。在此基础上，设计算法找到循环开始的节点（被两个不同祖先指向的节点）。

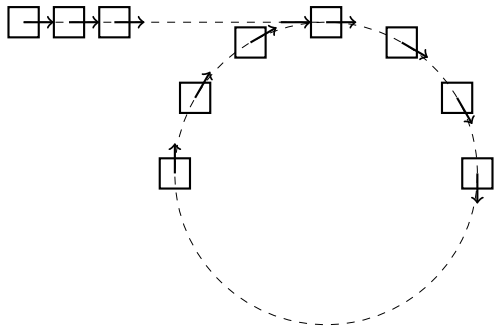


图 1.5: 带有循环的列表

第二章　二叉搜索树

数组和链表通常被认为是最基础的数据结构，其实它们并不简单。在某些系统中，数组是最基本的组件，甚至链表也可以由数组来实现（第 10.3 节 [4]）。另一方面，在函数式环境中，链表被作为最基本的组件来实现数组和其它更复杂的数据结构。

我们选择二叉搜索树作为数据结构中的“hello world”。乔 · 本特利（Jon Bentley）在《编程珠玑》[2] 一书中，讨论了如何统计一段文字中各单词出现的次数。下面的例子程序给出了一个解法。

```
void wordcount(Input in) {
    bst<string, int> map;
    while string w = read(in) {
        map[w] = if map[w] == null then 1 else map[w] + 1
    }
    for var (w, c) in map {
        print(w, ":", c)
    }
}
```

我们可以运行下面的命令进行统计：

```
$ cat bbe.txt | wordcount > wc.txt
```

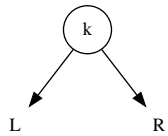
这里的 map 是用二叉搜索树实现的字典数据结构。我们用单词作为 key，用单词出现的次数作为值。这个程序运行快速，展示了二叉搜索树的强大功能。在详细介绍之前，我们先来了解一下二叉树。

2.1　定义

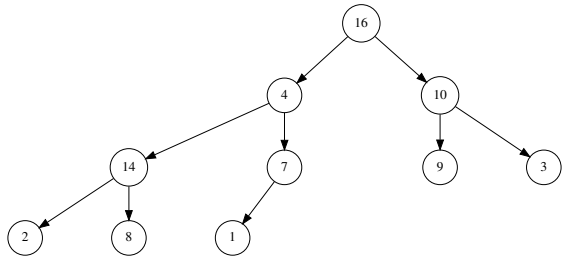
二叉树是一种递归的数据结构，一棵二叉树：

- 或者为空，
- 或者包含三个部分：一个元素和左右两个分支，这两个分支也都是二叉树。

左右分支也被称为左子树和右子树，或统称为孩子。我们也可以说一棵树由若干节点构成。节点中的值可以是任何类型或为空。如果一个节点的左右子树都为空，我们称之为叶子节点，否则称为分支节点。



(a) 二叉树的结构



(b) 一棵二叉树

图 2.1: 二叉树的结构和例子

二叉搜索树是一种特殊的二叉树，它的值可以进行比较[1]，并且满足下面的条件：

- 对于任何节点，所有左侧分支的值都小于本节点的值，
- 本节点的值小于所有右侧分支的值。

图2.2展示了一棵二叉搜索树。和图2.1比较，可以看到节点组织方式的不同。一棵二叉树的值可以是任意类型，而二叉搜索树要求它的值必须能进行比较[2]。为了强调这种区别，我们特别称二叉搜索树的的值为键 (key)，把节点存储的其他数据信息称为值 (value)。

2.2 数据组织

根据二叉搜索树的定义，我们可以用图2.3来描绘数据的组织结构。一个节点包含一个键和一些可选的额外数据。接下来是两个指向左右子树的两个引用。为了方便地从一个节点上溯到祖先，也可以存储一个指向父节点的引用。

简单起见，我们会忽略额外的存储数据。本章附录给出了一个例子定义。在函数式环境中，一般不使用引用或指针来进行回溯，而通常以自顶向下的递归来设计算法。以下是一个函数式的定义：

[1]广义的可比较，例如大小，先后、包含等序关系。本章中的“小于”及其符号 $<$ 是抽象的比较。

[2]只要能进行抽象的“小于”和“等于”比较就足够了。

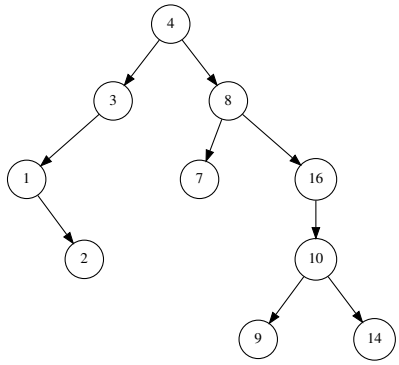


图 2.2: 二叉搜索树的例子

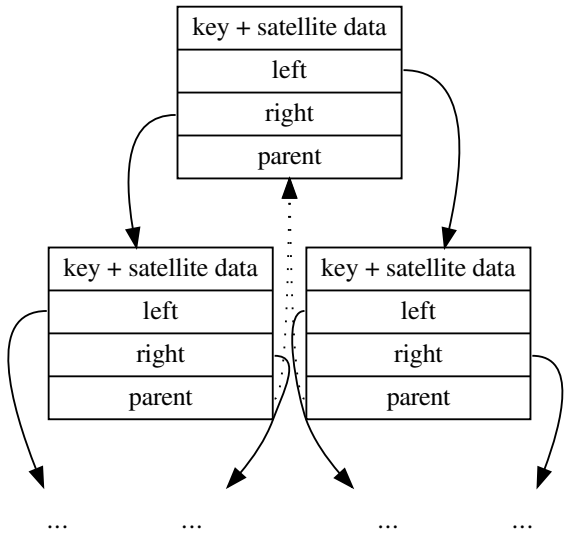


图 2.3: 带有父节点引用的数据组织

```
data Tree a = Empty
            | Node (Tree a) a (Tree a)
```

2.3 插入

当向二叉搜索树插入一个键 k（和相关的数据）时，我们需要确保树中的元素仍然是有序的。为此，我们可以设计如下的插入策略：

- 如果树为空，创建一个元素为 k 的叶子节点；
- 如果 k 小于根节点中的元素，将它插入到左子树中；
- 否则，将 k 插入到右子树中。

这里存在一个特殊情况：当 k 等于根节点中的元素时，说明它已经存在了。我们可以覆盖掉以前的数据，或者把新数据添加在后面，也可以跳过不做任何处理。简单起见，我们忽略这一情况。插入算法是递归的，它十分简单。可以定义为如下的函数：

$$\begin{aligned} insert(\varnothing, k) &= Node(\varnothing, k, \varnothing) \\ insert(Node(T_l, k, T_r), k) &= \begin{cases} k < k' : & Node(insert(T_l, k), k', T_r) \\ otherwise : & Node(T_l, k', insert(T_r, k)) \end{cases} \end{aligned} \tag{2.1}$$

当节点不为空时，T_l、T_r、k' 分别是它的左右子树和键。函数 $Node(l, k, r)$ 用左右子树和键来构造一个节点。符号 $\varnothing$ 表示空或 NIL。它是大数学家安德烈 · 韦伊引入的挪威语字母，用来表示空集。下面是相应的例子程序：

```
insert Empty k = Node Empty k Empty
insert (Node l x r) k | k < x = Node (insert l k) x r
                      | otherwise = Node l x (insert r k)
```

这一例子程序使用了模式匹配（pattern matching）特性。本章附录给出了另一个不使用此特性的例子。插入算法也可以不使用递归，而纯用迭代实现：

```
1:  function INSERT(T, k)
2:      root ← T
3:      x ← CREATE-LEAF(k)
4:      parent ← NIL
5:      while T ≠ NIL do
6:          parent ← T
7:          if k < KEY(T) then
8:              T ← LEFT(T)
9:          else
10:             T ← RIGHT(T)
```

```
11:     PARENT(x) ← parent
12:     if parent = NIL then                                  ▷ T 为空
13:         return x
14:     else if k < KEY(parent) then
15:         LEFT(parent) ← x
16:     else
17:         RIGHT(parent) ← x
18:     return root

19: function CREATE-LEAF(k)
20:     x ← EMPTY-NODE
21:     KEY(x) ← k
22:     LEFT(x) ← NIL
23:     RIGHT(x) ← NIL
24:     PARENT(x) ← NIL
25:     return x
```

这一实现虽然没有函数式算法简洁，但执行速度更快，并且可以处理深度很大的树。

2.4 遍历

遍历是指依次访问二叉树中的每个元素。有三种遍历方法，分别是前序遍历、中序遍历、后序遍历。它们是按照访问根节点和子节点的先后顺序命名的。

- 前序遍历：先访问**根节点**，然后访问左子树，最后访问右子树；
- 中序遍历：先访问左子树，然后访问**根节点**，最后访问右子树；
- 后序遍历：先访问左子树，然后访问右子树，最后访问**根节点**。

所有的“访问”操作都是递归的。**先**访问根后访问子分支称为**先序**，在访问左右分支的**中间**访问根称为**中序**，先访问子分支**后**访问根称为**后序**。对于图2.2中的二叉树，三种遍历的结果分别如下：

- 前序遍历：4, 3, 1, 2, 8, 7, 16, 10, 9, 14
- 中序遍历：1, 2, 3, 4, 7, 8, 9, 10, 14, 16
- 后序遍历：2, 1, 3, 7, 9, 14, 10, 16, 8, 4

特别地，中序遍历会按照从小到大的顺序输出元素。二叉搜索树的定义保证了这一性质。我们把相应的证明留作练习。中序遍历的算法可以描述为：

- 如果树为空，返回；
- 否则先中序遍历左子树，然后访问根节点，最后再中序遍历右子树。

这一描述本身是递归的。我们可以进一步定义一个 map 函数，按照中序遍历的顺序将函数 f 应用的每个元素上，从而映射成另一棵同构的树。

$$\begin{aligned} map(f, \varnothing) &= \varnothing \\ map(f, Node(T_l, k, T_r)) &= Node(map(f, T_l), f(k), map(f, T_r)) \end{aligned} \tag{2.2}$$

如果只访问并操作节点上的值，而无需创建另外一棵树，我们可以将这一算法实现如下：

```
1: function In-Order-Traverse(T, f)
2:     if T ≠ NIL then
3:         In-Order-Traverse(Left(T, f))
4:         f(Key(T))
5:         In-Order-Traverse(Right(T, f))
```

我们也可以修改 map 函数，通过中序遍历将一棵二叉搜索树转化为一个有序序列：

$$\begin{aligned} toList(\varnothing) &= [\,] \\ toList(Node(T_l, k, T_r)) &= toList(T_l) \mathbin{+\!\!+} [k] \mathbin{+\!\!+} toList(T_r) \end{aligned} \tag{2.3}$$

我们据此可以得到一个排序的方法：先把一个无序的列表转化为一个二叉搜索树，然后再用中序遍历把树转换回列表。该方法被称为“树排序”。记待排序列表为 $X = [x_1, x_2, x_3, ..., x_n]$。

$$sort(X) = toList(fromList(X)) \tag{2.4}$$

我们也可以写成函数组合 [8] 的形式：

$$sort = toList \circ fromList$$

其中函数 $fromList$ 不断地将元素从列表中插入到一棵树中，它可以递归地定义如下：

$$\begin{aligned} fromList([\,]) &= \varnothing \\ fromList(X) &= insert(fromList(X'), x_1) \end{aligned}$$

如果列表为空，则产生的树也是空；否则它把第一个元素 x_1 插入树中，然后递归地插入剩余元素 $X' = [x_2, x_3, ..., x_n]$。通过使用列表叠加 [7]（详见附录 A.6），我们也可以将 $fromList$ 定义为：

$$fromList(X) = \text{fold}_l(insert, \varnothing, X) \tag{2.5}$$

我们也可以进一步把它简写为柯里化的形式 [9]（也称为部分应用）从而省略掉参数 X：

$$fromList = \text{fold}_l\ \ insert\ \varnothing$$

练习 2.1

1. 给定如下前序遍历和中序遍历的结果，请重建出二叉树，并给出后序遍历的结果。
   - 前序遍历结果：1, 2, 4, 3, 5, 6
   - 中序遍历结果：4, 2, 1, 5, 3, 6
   - 后序遍历结果：?
2. 归纳前一题的规律，编程实现从前序遍历和中序遍历的结果重建二叉树。
3. 证明对二叉搜索树进行中序遍历可以将全部元素按照从小到大的顺序输出。
4. 对于 n 个元素，树排序的算法复杂度是什么？

2.5 搜索

由于二叉搜索树中的元素是按序递归存储的，它可以方便地支持各种搜索。这也是人们将其命名为“搜索树”的原因。有三种不同类型的搜索：1）在树中查找一个键；2）寻找最大或最小元素；3）查找某一元素的前驱（上一个）或后继（下一个）元素。

2.5.1 查找

二叉搜索树的定义使得它非常适合自顶向下的查找。可以按照下面的方法在树中查找元素 k：

- 如果树为空，结束查找，k 不存在；
- 如果根节点元素等于 k，结束查找。结果存储在根节点中；
- 如果 k 小于根节点元素，在左子树中递归查找；
- 否则，在右子树中递归查找。

我们可以定义递归的 $lookup$ 函数来实现这一算法：

$$\begin{aligned} lookup(\varnothing, x) &= \varnothing \\ lookup(Node(T_l, k, T_r), x) &= \begin{cases} k = x: & T \\ x < k: & lookup(T_l, x) \\ otherwise: & lookup(T_r, x) \end{cases} \end{aligned} \tag{2.6}$$

这一函数返回查找到的节点，如果没有找到就返回空。我们也可以返回节点内存储的值。这时可以使用 $Maybe$ 类型（也叫作 `Optional<T>`）来处理未找到的情况。例如：

```
lookup Empty _ = Nothing
lookup t@(Node l k r) x | k == x = Just k
                        | x < k = lookup l x
                        | otherwise = lookup r x
```

如果二叉树很平衡，大多数中间节点都有非空的左右分支（我们将在第四章给出平衡的定义），对于 n 个元素的二叉树，搜索算法的性能为 $O(\lg n)$。如果二叉树很不平衡，最坏情况下，查找的时间会退化到 $O(n)$。如果记树的高度为 h，则查找算法的性能可以表示成 $O(h)$ 的形式。

搜索算法也可以不使用递归来实现：

```
1: function SEARCH(T, x)
2:     while T ≠ NIL and KEY(T) ≠ x do
3:         if x < KEY(T) then
4:             T ← LEFT(T)
5:         else
6:             T ← RIGHT(T)
7:     return T
```

2.5.2 最小和最大元素

在二叉搜索树中，较小的元素总是位于左侧分支，而较大的元素总是位于右侧分支。可以利用这一特性来定位最大和最小元素。为了找到最小元素，我们可以不断向左侧前进，直到左侧分支为空。对称地，我们可以通过不断向右侧前进找到最大元素。

$$\begin{aligned} min(Node(\varnothing, k, T_r)) &= k \\ min(Node(T_l, k, T_r)) &= min(T_l) \end{aligned} \tag{2.7}$$

$$\begin{aligned} max(Node(T_l, k, \varnothing)) &= k \\ max(Node(T_l, k, T_r)) &= max(T_r) \end{aligned} \tag{2.8}$$

这两个函数的性能都是 $O(h)$，其中 h 是树的高度。

2.5.3 前驱和后继

有时需要把二叉搜索树当作通用容器，使用迭代器进行遍历。例如从最小的元素开始，逐一向前移动到最大元素，或者按需先后移动。下面的例子程序升序输出树中的元素：

```
void printTree (Node<T> t) {
    for (var it = Iterator(t), it.hasNext(); it = it.next()) {
```

```
        print(it.get(), ", ");
    }
}
```

这就需要查找一个给定节点的前驱或后继元素。x 的后继定义为全部满足 $x < y$ 中的最小的一个 y。如果 x 的右子树不为空，则右子树中的最小值就是后继。图2.4中 8 的后继元素为 9，它是 8 的右子树中的最小值。如果 x 的右子树为空，我们需要向上回溯，找到最近的一个祖先，使得该祖先的左子树也是 x 的祖先。在图2.4中，元素 2 所在的节点没有右侧分支，我们向上回溯一步找到元素 1，但是 1 没有左侧分支，因此需要继续向上查找，这次我们到达了元素 3 所在的节点。而 3 的左子树也是 2 的祖先。至此，我们找到了 2 的后继元素 3。

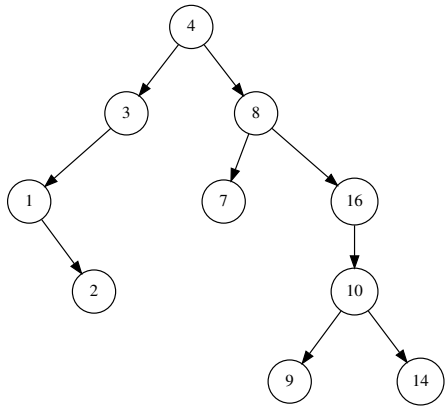


图 2.4: 8 的后继为其右侧分支的最小值 9；为了获得 2 的后继，首先向上找到 1，它没有左子树，所以继续向上找到 3，3 的左子树也是 2 的祖先，故而后继为 3。

如果沿着父节点引用一直回溯到了根节点，但是仍然没有找到位于右侧的祖先，这说明 x 没有后继（它是树中最后一个元素）。下面的算法实现了后继的查找：

```
1: function SUCC(x)
2:     if RIGHT(x) ≠ NIL then
3:         return MIN(RIGHT(x))
4:     else
5:         p ← PARENT(x)
6:         while p ≠ NIL and x = RIGHT(p) do
7:             x ← p
8:             p ← PARENT(p)
9:         return p
```

当 x 没有后继时，这一算法返回 NIL。寻找前驱元素的算法与此对称：

```
1: function PRED(x)
2:     if LEFT(x) ≠ NIL then
3:         return MAX(LEFT(x))
4:     else
5:         p ← PARENT(x)
6:         while p ≠ NIL and x = LEFT(p) do
7:             x ← p
8:             p ← PARENT(p)
9:         return p
```

似乎很难找到纯函数式算法实现前驱和后继的查找。这主要是因为在缺少指向父节点的引用[3]。一种折衷的方案是在遍历树的时候，留下一些“面包屑”作为标记。用以将来回溯甚至重建整棵树。这种同时包含树和“面包屑”信息的数据结构称为 zipper([10] 最后一章)。

查找前驱和后继的初衷是“作为一个通用容器，遍历树中的全部元素”。而在纯函数式环境中，我们通常用 map 函数中序遍历所有元素。前驱和后续的查找，仅在命令式环境中才有意义。另外一个这样仅在命令式环境中才有引起关注的例子是红黑树中元素的删除 [5]。

练习 2.2

1. 使用 PRED 和 SUCC 实现一个二叉搜索树的迭代器。用它遍历一棵含有 n 个元素的树的复杂度是什么?
2. 下面程序可以遍历一个区间 $[a, b]$ 内的元素:
   ```
   for_each (m.lower_bound(12), m.upper_bound(26), f);
   ```
   试用纯函数式的方法解决这一问题

2.6 删除

在二叉搜索树中删除元素需要额外的处理。我们必须保证删除后树的有序性质不能被破坏：对于任何节点，所有左侧分支的元素仍然小于节点中的元素，并且所有右侧分支的元素仍然大于节点中的元素。而删除节点会破坏这一性质。

从二叉搜索树中删除节点 x 的方法如下 [6]:

- 如果 x 没有非空子树（叶子）或者只有一棵非空子树，直接将 x “切下”;
- 否则，x 有棵非空子树，我们用其右子树中的最小值 y 替换掉 x，然后将原先的 y “切掉”。

[3] ML 或 OCaml 中有 `ref` 引用概念，这里我们限于纯函数式环境。

这一简洁的方法利用了这样一条特性：右子树中的最小值不可能有两个非空子树。所以上面的第二种情形转化为第一种情况，因而可以直接将原最小值节点“切掉”。

图2.5、2.6、2.7描述了删除时的各种情况。

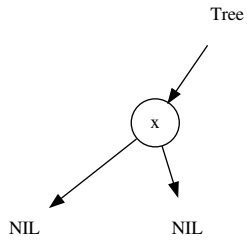
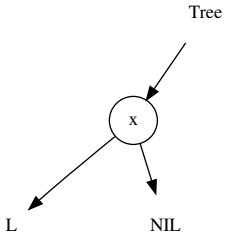
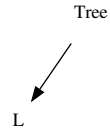


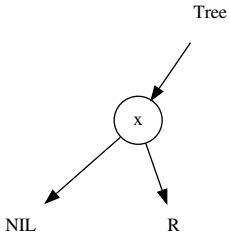
图 2.5: 叶子节点 x 可以直接“切下”



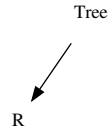
(a) 删除 x 前



(b) 删除 x 后。x 被“切掉”并由其左侧分支代替

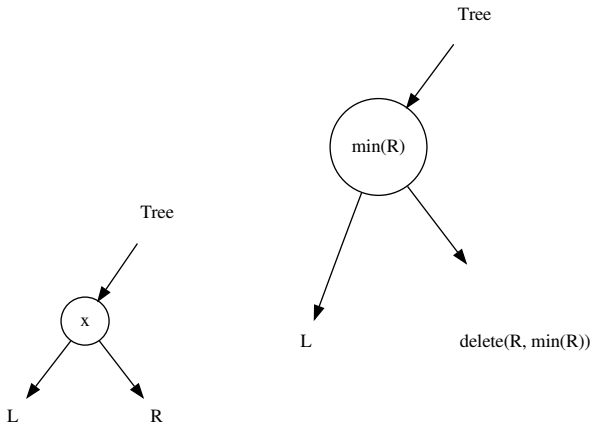


(c) 删除 x 前



(d) 删除 x 后。x 被“切掉”并由其右侧分支代替

图 2.6: 删除只有一个非空子分支的节点



(a) 删除 x 前　　(b) 删除 x 后。x 被替换为右侧分支中的被“切下”的最小值

图 2.7: 删除有两个非空分支的节点

根据这个思路，我们定义下面的 $delete$ 函数:

$$\begin{aligned} delete(\varnothing, x) &= \varnothing \\ delete(Node(T_l, k, T_r), x) &= \begin{cases} x < k: & Node(delete(T_l, x), k, T_r) \\ x > k: & Node(T_l, k, delete(T_r, x)) \\ x = k: & del(T_l, T_r) \end{cases} \end{aligned} \tag{2.9}$$

算法先通过序关系找到待删除节点，然后调用 del 函数处理，del 会根据情况递归调用 $delete$ 以删除右子树中的最小值。

$$\begin{aligned} del(\varnothing, T_r) &= T_r \\ del(T_l, \varnothing) &= T_l \\ del(T_l, T_r) &= Node(T_l, y, delete(T_r, y)) \end{aligned} \tag{2.10}$$

其中 $y = min(T_r)$ 是右子树中的最小元素。下面是相应的例子程序:

```
delete Empty _ = Empty
delete (Node l k r) x | x < k = Node (delete l x) k r
                      | x > k = Node l k (delete r x)
                      | otherwise = del l r
  where
    del Empty r = r
    del l Empty = l
    del l r = let k' = min r in Node l k' (delete r k')
```

如果树的高度为 h，则删除算法的复杂度为 $O(h)$。命令式算法需要在删除后，把父节点设置正确。下面的算法返回删除后树的根节点。

1: **function** DELETE(T, x)

```
2:    r ← T
3:    x' ← x                                                    ▷ save x
4:    p ← Parent(x)
5:    if Left(x) = NIL then
6:        x ← Right(x)
7:    else if Right(x) = NIL then
8:        x ← Left(x)
9:    else                                          ▷ neither children is empty
10:       y ← Min(Right(x))
11:       Key(x) ← Key(y)
12:       Copy other satellite data from y to x
13:       if Parent(y) ≠ x then                      ▷ y does not have left sub-tree
14:           Left(Parent(y)) ← Right(y)
15:       else                                ▷ y is the root of the right sub-tree
16:           Right(x) ← Right(y)
17:       if Right(y) ≠ NIL then
18:           Parent(Right(y)) ← Parent(y)
19:       Remove y
20:       return r
21:   if x ≠ NIL then
22:       Parent(x) ← p
23:   if p = NIL then                                          ▷ remove the root
24:       r ← x
25:   else
26:       if Left(p) = x' then
27:           Left(p) ← x
28:       else
29:           Right(p) ← x
30:   Remove x'
31:   return r
```

假定待删除的节点 x 不为空。算法首先记录下树的根节点、待删除的节点和它的父节点。如果 x 的任一分支为空，算法直接将 x “切掉”。否则，如果两个子分支都不为空，我们需要先在右子树中找到最小值节点 y。用 y 替换掉 x 中的值，同时将附加数据也替换过去。最后将原先的 y “切掉”。我们还需要处理 y 是 x 右子树的根节点这一特殊情况。

此后还需要把之前保存的父节点重新设好。如果父节点为空，则说明要删除的节点是根节点。这种情况下，我们需要返回新的根。当父节点被设置好后，就可以安全

把 x 删除了。对于高度为 h 的树，这一算法的复杂度也是 $O(h)$。

练习 2.3

1. 当节点的两个分支都不为空时，存在一种对称的删除算法：用左子树的最大值替换待删除的节点，然后将此最大值的节点“切下”。编程实现这一算法。

2.7 随机构建

本章给出的所有算法的复杂度都依赖于树的高度 h。如果树非常不平衡，$O(h)$ 就会接近 $O(n)$，因而退化为线性复杂度。反之，如果树平衡，$O(h)$ 接近 $O(\lg n)$，算法的性能就会很好。

第四、五章将介绍两种保证二叉搜索树的平衡的方法。这里我们先给出一个简单的方法（[4] 第 265-268 页）：可以通过随机构建来减小不平衡性。也就是说，在构建二叉搜索树前，先通过随机函数打乱元素的次序，然后再依次插入。

练习 2.4

1. 编程实现随机构建二叉搜索树。
2. 如何在一棵二叉树中找到“距离最远”的两个节点?

2.8 附录：例子代码

包含父节点引用的二叉搜索树的例子定义：

```
data Node<T> {
    T key
    Node<T> left
    Node<T> right
    Node<T> parent

    Node(T k) = Node(null, k, null)

    Node(Node<T> l, T k, Node<T> r) {
        left = l, key = k, right = r
        if (left ≠ null) then left.parent = this
        if (right ≠ null) then right.parent = this
    }
}
```

不使用模式匹配的递归插入算法：

```
Node<T> insert (Node<T> t, T x) {
    if (t == null) {
        return Node(null, x, null)
    } else if (t.key < x) {
```

```
        return Node(insert(t.left, x), t.key, t.right)
    } else {
        return Node(t.left, t.key, insert(t.right, x))
    }
}
```

消除递归的查找算法：

```
Optional<Node<T>> lookup (Node<T> t, T x) {
    while (t ≠ null and t.key ≠ x) {
        if (x < t.key) {
            t = t.left
        } else {
            t = t.right
        }
    }
    return Optional(t);
}
```

迭代寻找最小元素：

```
Optional<Node<T>> min (Node<T> t) {
    while (t ≠ null and t.left ≠ null) {
        t = t.left
    }
    return Optional(t);
}
```

寻找给定节点的后继：

```
Optional<Node<T>> succ (Node<T> x) {
    if (x == null) {
        return Optional.None
    } else if (x.right ≠ null) {
        return min(x.right)
    } else {
        p = x.parent
        while (p ≠ null and x == p.right) {
            x = p
            p = p.parent
        }
        return Optional(p);
    }
}
```

第三章　插入排序

插入排序是一种简单直观的排序算法[1]。在第一章中，我们给出了它的简明定义：对于一组可比较的元素，我们不断从中取出元素，按序将其插入到一个列表中。由于每次插入都需要线性时间，排序的复杂度为 $O(n^2)$，其中 n 是元素的个数。插入排序的性能不如一些分而治之的排序算法，例如快速排序和归并排序。尽管如此，我们仍然能在现代软件中找到插入排序的应用。在快速排序的实现中，通常在数据集较小的时候，回退到插入排序。

3.1　简介

扑克游戏中的抓牌环节非常形像地描述了插入排序的思想 ([4] 第 15 - 19 页)。考虑从一副洗好的牌中不断抓牌，并按序理好的过程。任何时候，人们手中的牌都是有序的。每当抓到一张新牌，就按照牌的点数，插入到合适的位置。如图3.1所示。根据这一思路，我们可以这样实现插入排序：

图 3.1: 将草花 8 插入到一手牌中

```
1: function SORT(A)
2:     S ← NIL
3:     for each a ∈ A do
4:         INSERT(a, S)
```

[1] 忽略冒泡排序算法

```
5:    return S
```

这一实现将排序的结果存储在新数组 S 中，也可以复用原数组的空间进行就地排序：

```
1: function SORT(A)
2:     for i ← 2 to |A| do
3:         ordered insert A[i] to A[1...(i − 1)]
```

其中索引 i 的范围是从 1 到 $n = |A|$。只含有一个元素的子数组 $[A[1]]$ 是已序的，因此我们从第二个元素开始插入。当处理第 i 个元素时，所有 i 之前的元素是已序的。我们不断将未排序的元素插入，如图3.2所示。

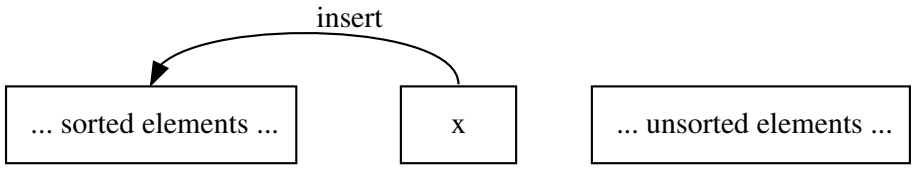


图 3.2: 不断将元素插入已序部分

3.2 插入

第一章给出了列表的插入算法。对于数组，也可以通过逐一检查找到插入位置。检查可以从左向右或者从右向左进行。下面的实现是从右向左进行检查的：

```
1: function SORT(A)
2:     for i ← 2 to |A| do                          ▷ Insert A[i] to A[1...(i − 1)]
3:         x ← A[i]                                  ▷ 将 A[i] 保存到 x
4:         j ← i − 1
5:         while j > 0 and x < A[j] do
6:             A[j + 1] ← A[j]
7:             j ← j − 1
8:         A[j + 1] ← x
```

由于数组是连续存储的，在某一位置插入元素是一个代价较高的操作。若在第 i 个位置插入元素 x，需要把 i 后面的所有元素（包括 $A[i+1]$、$A[i+2]$……）都向右移动一个位置。将第 i 个位置空出以放入 x。如图3.3所示。

数组的长度为 n，若比较 x 和前 i 个元素后，定位到了插入位置。接下来需要将剩余的 $n - i + 1$ 的元素向后移动，再将 x 放入第 i 个位置。整体上看，我们相当于从左向右遍历了整个数组。另一方面，如果从右向左处理，则需要检查 $n - i + 1$ 个元素，并执行相同数量的移动操作。我们也可以定义一个单独的 INSERT() 函数，并在循环中调用。无论是从左向右或从右向左处理，插入操作都需要线性时间，因此插

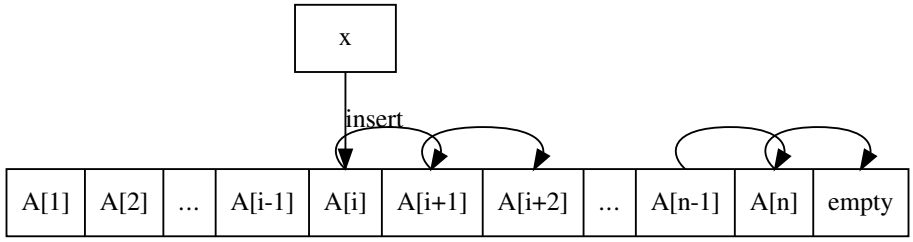


图 3.3: 将元素 x 插入数组 A 中的第 i 个位置

入排序的总体复杂度为 $O(n^2)$，其中 n 是元素的个数。

练习 3.1

1. 实现从左向右处理的插入操作。
2. 定义单独的插入函数以实现插入排序。

3.3 二分查找

在玩扑克牌的时候，人们并不是逐一比较找到插入位置的。我们之所以能快速定位，是因为手中的牌在任何时刻都是已序的。二分查找是一种在已序序列中快速定位的方法。

```
1: function SORT(A)
2:     for i ← 2 to |A| do
3:         x ← A[i]
4:         p ← BINARY-SEARCH(x, A[1...(i − 1)])
5:         for j ← i down to p do
6:             A[j] ← A[j − 1]
7:         A[p] ← x
```

二分查找时，数组中的片断 $A[1...(i-1)]$ 是有序的。不失一般性，设其为单调增(可以定义抽象的 $\leq$)。我们需要找到一个位置 j 使得 $A[j-1] \leq x \leq A[j]$。我们先用 x 和中间位置的元素 $A[m]$ 比较，其中 $m = \lfloor \frac{i}{2} \rfloor$。如果 $x < A[m]$，则递归地在前一半序列中二分查找；否则查找后一半序列。由于每次都排除掉一半元素，二分查找需要 $O(\lg i)$ 的时间定位到插入点。

```
1: function BINARY-SEARCH(x, A)
2:     l ← 1, u ← 1 + |A|
3:     while l < u do
4:         m ← ⌊(l + u)/2⌋
5:         if A[m] = x then
```

```
            return m                                   ▷ 重复元素
        else if A[m] < x then
            l ← m + 1
        else
            u ← m
    return l
```

这一改进并不能提高插入排序的整体复杂度，结果仍然是 $O(n^2)$。逐一比较的插入排序需要 $O(n^2)$ 次比较和 $O(n^2)$ 次移动；使用二分查找后，比较次数提高到了 $O(n \lg n)$，但移动次数还是 $O(n^2)$。

练习 3.2

1. 使用递归实现二分查找。

3.4 列表

二分查找将搜索时间降低到 $O(n \lg n)$，但由于要依次移动数组中的元素，整体复杂度仍然是 $O(n^2)$。另一方面，使用列表存储元素时，一旦获取了插入位置的引用，插入操作本身是常数时间的。在第一章中，我们定义了如下的列表插入排序算法：

$$\begin{aligned} sort(\varnothing) &= \varnothing \\ sort(x : xs) &= insert(x, sort(xs)) \end{aligned} \tag{3.1}$$

或使用 $fold_l$ 的柯里化形式：

$$sort = fold_l(insert, \varnothing) \tag{3.2}$$

由于需要遍历，列表的 $insert$ 算法仍是线性时间的：

$$\begin{aligned} insert(x,\ \varnothing) &= [x] \\ insert(x,\ y : ys) &= \begin{cases} x \leq y : & x : y : ys \\ otherwise : & y : insert(x, ys) \end{cases} \end{aligned} \tag{3.3}$$

也可以不使用节点引用，而通过另一个索引数组来实现列表。对任何元素 $A[i]$，$Next[i]$ 保存了 $A[i]$ 之后下一个元素的索引。也就是说 $A[Next[i]]$ 是 $A[i]$ 的下一个元素。其中有两个特殊索引：对于列表的末尾元素 $A[m]$，定义 $Next[m] = -1$，表示其指向 NIL；此外定义 $Next[0]$ 指向列表的头部。利用索引数组，我们定义插入算法如下：

```
1: function INSERT(A, Next, i)
2:     j ← 0                                   ▷ Next[0] 指向表头
3:     while Next[j] ≠ −1 and A[Next[j]] < A[i] do
```

```
        j ← Next[j]
    Next[i] ← Next[j]
    Next[j] ← i

function SORT(A)
    n ← |A|
    Next = [1, 2, ..., n, −1]                    ▷ n + 1 个索引
    for i ← 1 to n do
        INSERT(A, Next, i)
    return Next
```

使用列表，尽管在引用位置进行插入只需要常数时间，但必须遍历才能找到插入位置。整体仍需要 $O(n^2)$ 次比较。与数组不同，列表不支持随机访问，不能利用二分查找提升定位速度。

练习 3.3

1. 使用索引数组，排序结果是一个重新排列的索引。给出一个方法，根据新的索引 $Next$，重新排列数组 A。

3.5 二叉搜索树

我们遇到了一个困难境地：必须同时提高查找和插入的速度，仅提高其中之一仍然是 $O(n^2)$ 的复杂度。一方面，我们希望用二分查找把比较次数降低到 $O(\lg n)$；另一方面，需要改变数据结构，因为数组不支持在指定位置以常数时间插入元素。在第二章中，我们介绍了二叉搜索树。它天然就支持二分查找。一旦定位到插入位置，我们可以用常数时间插入新节点。

```
function SORT(A)
    T ← ∅
    for each x ∈ A do
        T ← INSERT-TREE(T, x)
    return TO-LIST(T)
```

第二章给出了 INSERT-TREE() 和 TO-LIST() 的定义。平均情况下，树排序的复杂度为 $O(n \lg n)$，其中 n 是元素的个数。这达到了基于比较的排序算法时间下限（[12] 第 180-193 页、[4] 第 167 页）。但在最坏情况下，当树极度不平衡时，其性能会下降到 $O(n^2)$。

3.6 小结

很多情况下，插入排序常作为第一个排序算法被介绍。它简单直观，但性能是平方级别的。插入排序不仅出现在教科书中，也出现在快速排序的工程实现中：在小数据集时回退到插入排序以抵消递归的代价。

第四章　红黑树

第二章的例子使用二叉搜索树来统计文章中每个词的出现次数。能否是用二叉搜索树处理通讯录，用来查询联系人的电话呢？如下面的例子代码所示：

```
void addrBook(Input in) {
    bst<string, string> dict
    while (string name, string addr) = read(in) {
        dict[name] = addr
    }
    loop {
        string name = read(console)
        var addr = dict[name]
        if (addr == null) {
            print("not found")
        } else {
            print("address: ", addr)
        }
    }
}
```

但这个方法性能不佳，尤其是搜索诸如 Zara、Zed、Zulu 等姓名时更加明显。通讯录是按照字典顺序排列的。如果依次把自然数 $1, 2, 3, ..., n$ 插入二叉搜索树，就会得到图4.1中的结果。这是一棵极不平衡的二叉树。对于高为 h 的二叉搜索树，查找的复杂度为 $O(h)$。如果树比较平衡，我们就能够达到 $O(\lg n)$ 的性能。但在这一极端情况下，查找的性能退化为 $O(n)$。几乎等同于列表扫描。

练习 4.1

1. 对于较大的通讯录，为了加快构建速度，可以使用两个并发的任务：一个从头部向后，另外一个从后向前读取。当两个任务相遇时结束。这样构建出的二叉搜索树是什么样子的？如果把通讯录分成更多片断，使用多任务会得到什么结果？

2. 参考图4.2，找出更多的不平衡情况。

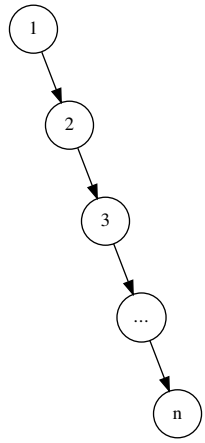


图 4.1: 不平衡的树

4.0.1 平衡

为了避免这种极不平衡的情况，可以将输入序列打乱（[4]12.4 节）。但这种方法有一定的局限性，如果序列是用户交互输入的，就无法打乱了。人们找到了一些解决平衡性的方法，它们大多依赖二叉树的旋转操作。旋转操作可以在改变树结构的同时，保持元素间顺序不变。这一章介绍红黑树。它是一种被广泛使用的自平衡二叉搜索树。下一章介绍另外一种自平衡树——AVL 树。第 8 章还会介绍伸展树，它能够随着操作，逐步把树变得平衡。

4.0.2 树旋转

树旋转在保持中序遍历结果不变的情况下，改变树的结构。存在多个不同的二叉树对应到一个特定的中序遍历顺序。图4.3描述了旋转操作。

旋转操作可以通过模式匹配来定义：

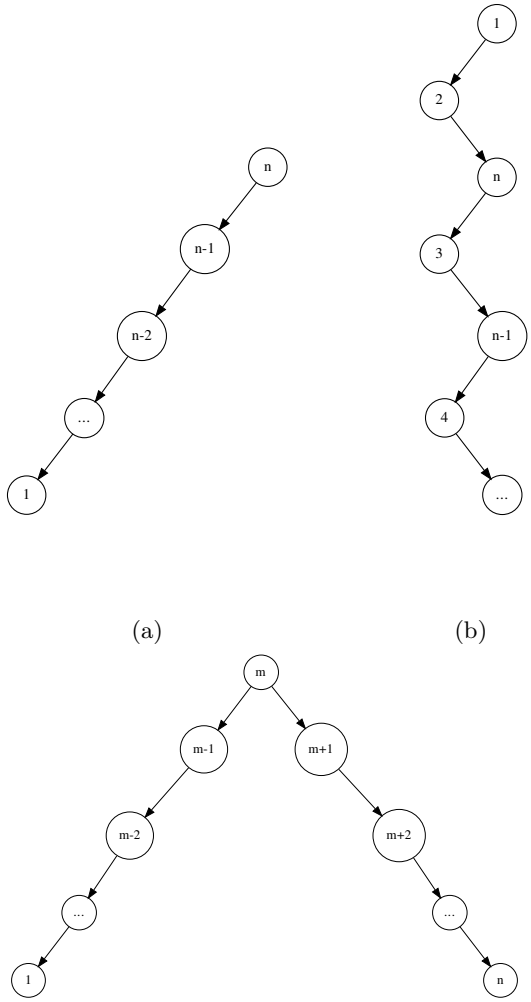
$$\begin{aligned} rotate_l\ ((a, x, b), y, c) &= (a, x, (b, y, c)) \\ rotate_l\ T &= T \end{aligned} \tag{4.1}$$

和

$$\begin{aligned} rotate_r\ (a, x, (b, y, c)) &= ((a, x, b), y, c)) \\ rotate_r\ T &= T \end{aligned} \tag{4.2}$$

如果模式没有匹配（例如两棵子树都为空），每个式子的第二行保持树不变。旋转操作也可以通过一系列步骤实现。我们需要将子树和父引用设置正确。在旋转时，我们传入根节点 T 和要旋转的子树 x：

1: **function** LEFT-ROTATE(T, x)
2: $p \leftarrow$ PARENT(x)



(a) (b)

(c)

图 4.2: 一些不平衡的二叉树

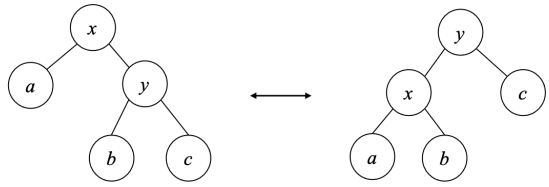


图 4.3: 树的左右旋转

```
    y ← RIGHT(x)                                  ▷ 设 y ≠ NIL
    a ← LEFT(x)
    b ← LEFT(y)
    c ← RIGHT(y)
    REPLACE(x, y)                                 ▷ 用 y 替换 x
    SET-SUBTREES(x, a, b)                         ▷ 令 a, b 为 x 的子树
    SET-SUBTREES(y, x, c)                         ▷ 令 x, c 为 y 的子树
    if p = NIL then                               ▷ 此前 x 是根节点
        T ← y
    return T
```

右旋 RIGHT-ROTATE 的实现是对称的，我们将其留作练习。REPLACE(x, y) 使用 y 替换 x：

```
function REPLACE(x, y)
    p ← PARENT(x)
    if p = NIL then                               ▷ x 是根节点
        if y ≠ NIL then PARENT(y) ← NIL
    else if LEFT(p) = x then
        SET-LEFT(p, y)
    else
        SET-RIGHT(p, y)
    PARENT(x) ← NIL
```

SET-SUBTREES(x, L, R) 将 L 设为 x 的左子树，R 设为右子树：

```
function SET-SUBTREES(x, L, R)
    SET-LEFT(x, L)
    SET-RIGHT(x, R)
```

它进一步调用 SET-LEFT 和 SET-RIGHT 完成子树的设置：

```
function SET-LEFT(x, y)
    LEFT(x) ← y
    if y ≠ NIL then PARENT(y) ← x


function SET-RIGHT(x, y)
    RIGHT(x) ← y
    if y ≠ NIL then PARENT(y) ← x
```

通过对比，可以看到模式匹配如何简化树旋转的实现。从这一点出发 Okasaki 在 1995 年实现了红黑树的纯函数式算法 [13]。

练习 4.2

1. 实现右旋 RIGHT-ROTATE 操作。

4.1 定义

红黑树是一种自平衡二叉搜索树 [14]。它是 2-3-4 树的等价形式[1]。通过对节点进行着色和旋转，红黑树可以高效地保持平衡。我们在二叉搜索树的定义上给节点赋予红、黑颜色。我们称一棵树为红黑树，如果它满足下面 5 条性质 [4]：

1. 节点的颜色为红色或黑色。
2. 根节点为黑色。
3. 所有叶节点（NIL）为黑色。
4. 如果一个节点为红色，则它的两个子节点都是黑色。
5. 从任一节点出发到所有叶子节点的路径上包含相同数量的黑色节点。

为什么这 5 条性质能保证红黑树的平衡性呢？关键在于：从根节点出发到达叶节点的所有路径中，最长路径不会超过最短路径两倍。性质 4 保证了不存在两个连续的红色节点。因此，最短的路径只含有黑色的节点。任何更长的路径一定含有红色节点。根据性质 5，从任何节点出发的所有的路径都含有相同数量的黑色节点，自然这条对于根节点也成立。这就最终保证了没有任何路径超过最短路径长度的两倍 [14]。图4.4的例子展示了一棵红黑树。

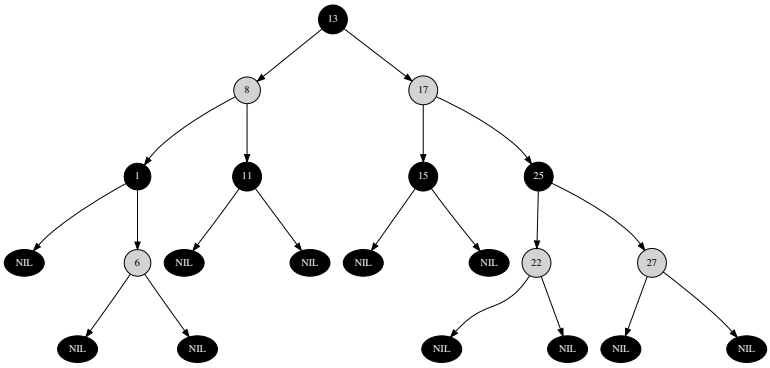


图 4.4: 红黑树

由于所有的 NIL 节点都是黑色的，我们通常将 NIL 节点隐藏不画出，如图4.4所示。所有不改变树结构的操作都和二叉搜索树相同，包括查找、最大、最小值等。只有插入和删除操作是特殊的。

[1]第 7 章，B 树。对于任一 2-3-4 树，都存在至少一棵红黑树，其元素顺序相同。

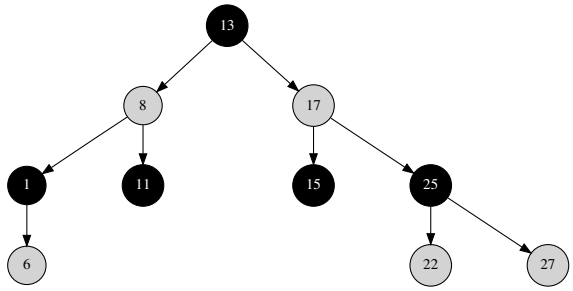


图 4.5: 隐藏 NIL 节点

下面的例子程序在二叉搜索树的基础上增加了颜色定义：

```
data Color = R | B
data RBTree a = Empty
              | Node Color (RBTree a) a (RBTree a)
```

练习 4.3

1. 证明含有 n 个节点的红黑树，其高度 h 不会超过 $2\lg(n+1)$。

4.2 插入

插入算法包含两个步骤：第一步和二叉搜索树相同，树可能会变得不再平衡；第二步修复红黑树的颜色性质。插入时，我们令新节点为红色。只要它不是根节点，除了第四条外的所有性质都可以满足。唯一的问题是可能引入两个相邻的红色节点，共有 4 种情况需要修复。Okasaki 发现它们具有统一的形式 [13]，如图 4.6所示。

四种情况都把红色向上移动一层。如果进行自底向上的递归修复，可能会把根节点染成红色。根据性质 2，最后需要把根节点变回黑色。利用模式匹配，我们定义 $balance$ 函数修复平衡。令节点的颜色变量为 $\mathcal{C}$，取值为黑色 $\mathcal{B}$ 或红色 $\mathcal{R}$。非空节点表达为一个四元组 $T = (\mathcal{C}, l, k, r)$，其中 l、r 是左右子树，k 是值。

$$\begin{array}{rcl} balance\ \mathcal{B}\ (\mathcal{R},(\mathcal{R},a,x,b),y,c)\ z\ d & = & (\mathcal{R},(\mathcal{B},a,x,b),y,(\mathcal{B},c,z,d)) \\ balance\ \mathcal{B},(\mathcal{R},a,x,(\mathcal{R},b,y,c))\ z\ d & = & (\mathcal{R},(\mathcal{B},a,x,b),y,(\mathcal{B},c,z,d)) \\ balance\ \mathcal{B}\ a\ x\ (\mathcal{R},b,y,(\mathcal{R},c,z,d)) & = & (\mathcal{R},(\mathcal{B},a,x,b),y,(\mathcal{B},c,z,d)) \\ balance\ \mathcal{B}\ a\ x\ (\mathcal{R},(\mathcal{R},b,y,c),z,d) & = & (\mathcal{R},(\mathcal{B},a,x,b),y,(\mathcal{B},c,z,d)) \\ balance\ T & = & T \end{array} \tag{4.3}$$

如果四种模式都不满足，最后一行保证此时不会改变树的形状。红黑树的插入算法定义如下：

$$insert\ T\ k = makeBlack\ (ins\ T\ k) \tag{4.4}$$

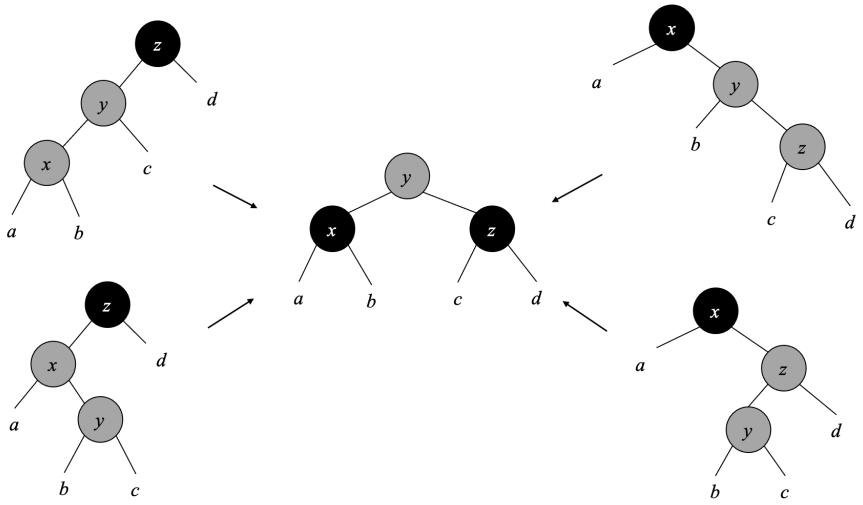


图 4.6: 插入后需要修复的四种情况

其中

$$
\begin{aligned}
ins\ \varnothing\ k &= (\mathcal{R}, \varnothing, k, \varnothing) \\
ins\ (\mathcal{C}, l, k', r)\ k &= \begin{cases} k < k': & balance\ \mathcal{C}\ (ins\ l\ k)\ k'\ r \\ k > k': & balance\ \mathcal{C}\ l\ k'\ (ins\ r\ k) \end{cases}
\end{aligned} \tag{4.5}
$$

如果树为空，我们为 k 创建一个红色叶子节点；否则，令树的左右分支和值分别为 l、r、k'。比较 k 和 k' 的大小，递归地将 k 插入到子树中。然后用 $balance$ 修复平衡性。最后强制把根节点染成黑色。

$$makeBlack\ (\mathcal{C}, l, k, r) = (\mathcal{B}, l, k, r) \tag{4.6}$$

下面是对应的例子程序：

```
insert t x = makeBlack $ ins t where
    ins Empty = Node R Empty x Empty
    ins (Node color l k r)
        | x < k     = balance color (ins l) k r
        | otherwise = balance color l k (ins r)
    makeBlack(Node _ l k r) = Node B l k r

balance B (Node R (Node R a x b) y c) z d =
                Node R (Node B a x b) y (Node B c z d)
balance B (Node R a x (Node R b y c)) z d =
                Node R (Node B a x b) y (Node B c z d)
balance B a x (Node R b y (Node R c z d)) =
                Node R (Node B a x b) y (Node B c z d)
balance B a x (Node R (Node R b y c) z d) =
                Node R (Node B a x b) y (Node B c z d)
balance color l k r = Node color l k r
```

我们略去了重复值的处理。如果要插入的值已经存在，我们可以覆盖或丢弃，还可以在节点中用一个列表存储相应的数据 ([4]，269 页)。图4.7给出了两棵红黑树。它们分别由序列 11, 2, 14, 1, 7, 5, 8, 4 和 1, 2, ..., 8 构建而成。第二个例子说明，即使序列已序，红黑树仍然保持平衡。

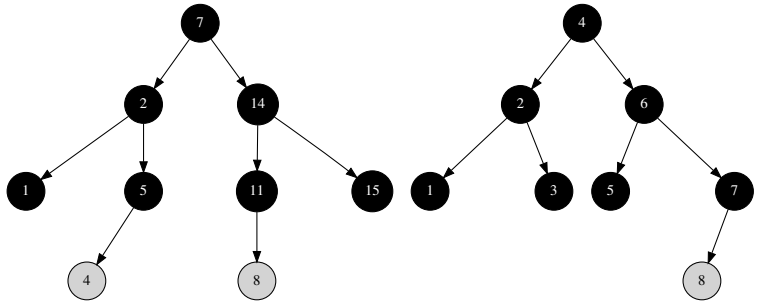


图 4.7: 插入产生的红黑树

算法自顶向下递归地进行插入和修复，对于高度为 h 的树，其复杂度为 $O(h)$。由于我们始终维护红黑树的颜色性质，h 和节点个数 n 呈对数关系。插入算法的复杂度为 $O(\lg n)$。

练习 4.4

1. 不使用模式匹配，分别检查四种情况实现 $insert$ 算法。

4.3 删除

红黑树的删除比插入复杂。也可以通过模式匹配和递归简化删除算法[2]的实现。我们还可以利用其它方式来达到删除的效果。例如，一次性构建一棵树，用于后继的多次查找 [5]。删除时在节点上加一个标记，当带有标记的节点超过 50% 时，用未标记的节点重建一棵树。删除也会破坏红黑树的性质，因此同样需要进行修复。问题只发生在删除黑色节点时，这会违反性质 5，使得某一路径上的黑色节点数目少于其它的路径。

我们可以引入“双重黑色”（[4]，290 页）节点来恢复第五条性质。一个这样的节点算作两个黑色节点。删除黑色节点 x 时，我们将黑色向上移动到父节点，或者向下移动到子树上。令接受黑色的节点为 y。如果 y 原来是红色，将其变为黑色；如果 y 原来是黑色，则变为“双重黑色”，记作 $\mathcal{B}^2$。下面的例子程序增加了双重黑色的定义：

```
data Color = R | B | BB
data RBTree a = Empty | BBEmpty
              | Node Color (RBTree a) a (RBTree a)
```

[2]实际上通过重用不变的部分重新构建了树。这一特性称作 persist

由于所有的空节点都是黑色，当将黑色移动到空节点时，其变为“双重黑色”空节点（BBEmpty 或加粗的 $\boldsymbol{\varnothing}$）。删除时，第一步和普通二叉搜索树相同；如果被删除节点是黑色的，接下来进行修复：

$$delete = makeBlack \circ del \tag{4.7}$$

这一定义是柯里化的。如果树中只有一个元素，删除后它变为空。为了处理这一情况，我们需要修改 $makeBlack$ 的定义如下：

$$\begin{aligned} makeBlack\ \varnothing &= \varnothing \\ makeBlack\ (\mathcal{C}, l, k, r) &= (\mathcal{B}, l, k, r) \end{aligned} \tag{4.8}$$

del 接受一棵树和要删除的元素 k：

$$\begin{aligned} del\ \varnothing\ k &= \varnothing \\ del\ (\mathcal{C}, l, k', r)\ k &= \begin{cases} k < k': & fixB^2(\mathcal{C}, (del\ l\ k), k', r) \\ k > k': & fixB^2(\mathcal{C}, l, k', (del\ r\ k)) \\ k = k': & \begin{cases} l = \varnothing: & (\mathcal{C} = \mathcal{B} \mapsto shiftB\ r, r) \\ r = \varnothing: & (\mathcal{C} = \mathcal{B} \mapsto shiftB\ l, l) \\ \text{否则}: & fixB^2(\mathcal{C}, l, k'', (del\ r\ k'')) \\ & \text{其中 } k'' = min(r) \end{cases} \end{cases} \end{aligned} \tag{4.9}$$

如果树为空，结果为 $\varnothing$；否则我们比较 k 和树中的 k'，如果 $k < k'$，我们递归地从左侧分支删除 k；如果 $k > k'$，则递归地从右侧删除。由于递归结果中可能含有双重黑色节点，需要应用 $fixB^2$ 进行修复。当 $k = k'$ 时，我们定位到了要删除的节点。如果任一子树为空，我们用另一子树替换掉当前节点。如果当前节点是黑色的，还需要将黑色移动到子树中。这段定义使用了麦卡锡形式 $(p \mapsto a, b)$，它相当于条件表达式：(if p then a else b)。如果两棵子树都不为空，我们将右子树中的最小值 $k'' = min(r)$ 切下，并用 k'' 替换 k。

为了保持黑色节点个数，$shiftB$ 将红色节点变为黑色，将黑色节点变为双重黑色。如果再次应用到双重黑色节点上，则变回黑色。

$$\begin{aligned} shiftB\ (\mathcal{B}, l, k, r) &= (\mathcal{B}^2, l, k, r) \\ shiftB\ (\mathcal{C}, l, k, r) &= (\mathcal{B}, l, k, r) \\ shiftB\ \varnothing &= \boldsymbol{\varnothing} \\ shiftB\ \boldsymbol{\varnothing} &= \varnothing \end{aligned} \tag{4.10}$$

下面是相应的例子程序（不包含双重黑色的修复部分）：

```
delete :: (Ord a) ⇒ RBTree a → a → RBTree a
delete t k = makeBlack $ del t k where
    del Empty _ = Empty
    del (Node color l k' r) k
```

```
        | k < k' = fixDB color (del l k) k' r
        | k > k' = fixDB color l k' (del r k)
        | isEmpty l = if color == B then shiftBlack r else r
        | isEmpty r = if color == B then shiftBlack l else l
        | otherwise = fixDB color l k'' (del r k'') where k''= min r
    makeBlack (Node _ l k r) = Node B l k r
    makeBlack _ = Empty

shiftBlack (Node B l k r) = Node BB l k r
shiftBlack (Node _ l k r) = Node B  l k r
shiftBlack Empty = BBEmpty
shiftBlack BBEmpty = Empty
```

函数 $fixB^2$ 通过旋转操作和重新染色消除双重黑色。双重黑色节点既可能是分枝节点，也可能是空节点 $\varnothing$。有三种情况：

情况 1：双重黑色的兄弟节点为黑色，并且该兄弟节点有一个红色子节点。可以通过旋转修复这种情况。共有四种子情况，全部可以变换到一种统一形式。如图A.1所示。

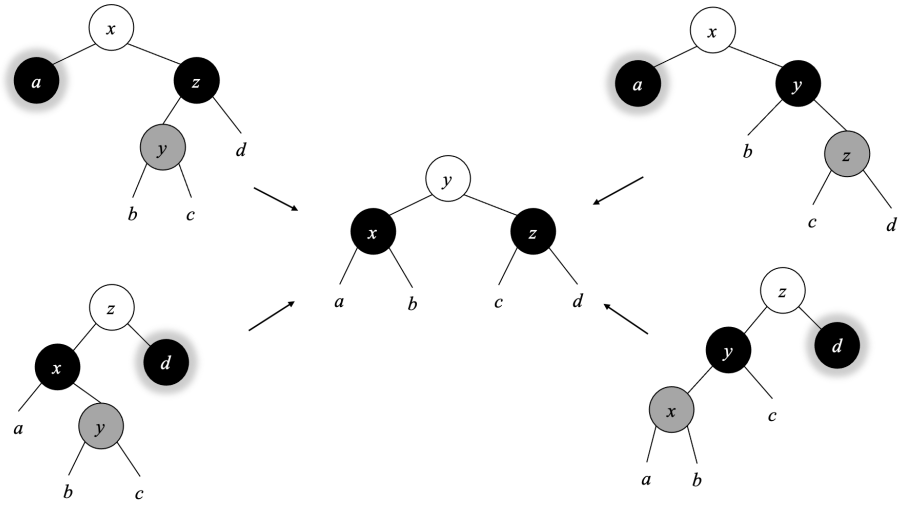


图 4.8: 4 种子情况可以修复为统一的形式

我们通过模式匹配来处理这四种子情况：

$$
\begin{aligned}
fixB^2\ \mathcal{C}\ a_{\mathcal{B}^2}\ x\ (\mathcal{B},(\mathcal{R},b,y,c),z,d) &= (\mathcal{C},(\mathcal{B},shiftB(a),x,b),y,(\mathcal{B},c,z,d)) \\
fixB^2\ \mathcal{C}\ a_{\mathcal{B}^2}\ x\ (\mathcal{B},b,y,(\mathcal{R},c,z,d)) &= (\mathcal{C},(\mathcal{B},shiftB(a),x,b),y,(\mathcal{B},c,z,d)) \\
fixB^2\ \mathcal{C}\ (\mathcal{B},a,x,(\mathcal{R},b,y,c))\ z\ d_{\mathcal{B}^2} &= (\mathcal{C},(\mathcal{B},a,x,b),y,(\mathcal{B},c,z,shiftB(d))) \\
fixB^2\ \mathcal{C}\ (\mathcal{B},(\mathcal{R},a,x,b),y,c)\ z\ d_{\mathcal{B}^2} &= (\mathcal{C},(\mathcal{B},a,x,b),y,(\mathcal{B},c,z,shiftB(d)))
\end{aligned}
\tag{4.11}
$$

其中 $a_{\mathcal{B}^2}$ 表示节点 a 是双重黑色，可以是分枝节点或 $\varnothing$。

情况 2：双重黑色节点的兄弟节点为红色。可以通过旋转，将其变换为情况 1 或 3。如图A.2所示。

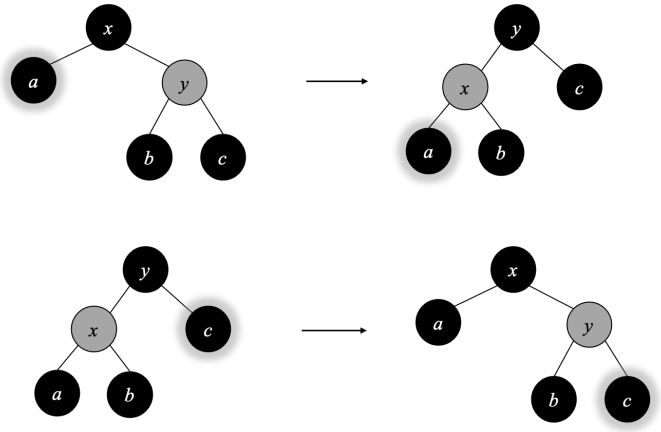


图 4.9: 双重黑色节点的兄弟节点为红色

我们在公式（4.11）的基础上增加情况 2 的修复：

$$
\begin{aligned}
& \ldots \\
fixB^2\ \mathcal{B}\ a_{\mathcal{B}^2}\ x\ (\mathcal{R}, b, y, c) &= fixB^2\ \mathcal{B}\ (fixB^2\ \mathcal{R}\ a\ x\ b)\ y\ c \\
fixB^2\ \mathcal{B}\ (\mathcal{R}, a, x, b)\ y\ c_{\mathcal{B}^2} &= fixB^2\ \mathcal{B}\ a\ x\ (fixB^2\ \mathcal{R}\ b\ y\ c)
\end{aligned}
\tag{4.12}
$$

情况 3：双重黑色的兄弟节点，该兄弟节点的两个子节点都是黑色。这种情况下，我们将兄弟节点染成红色，将双重黑色变回黑色，然后将双重黑色属性向上传递一层到父节点。如图A.3所示。

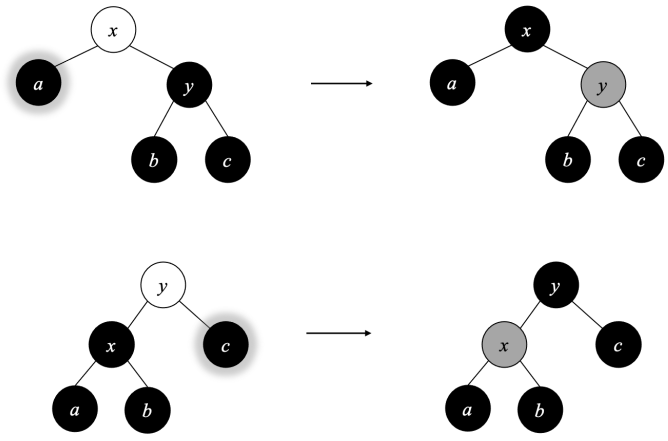


图 4.10: 将双重黑色向上传递

有两种对称情况：对于上方的情况，如果 x 是红色，则变为黑色，否则变为双重黑色；对于下方的情况，y 的变化与此类似。我们继续在式（4.12）的基础上增加此

种修复：

$$
\begin{array}{rcl}
 & \cdots & \\
fixB^2\ \mathcal{C}\ a_{\mathcal{B}^2}\ x\ (\mathcal{B}, b, y, c) & = & shiftB\ (\mathcal{C}, (shiftB\ a), x, (\mathcal{R}, b, y, c)) \\
fixB^2\ \mathcal{C}\ (\mathcal{B}, a, x, b)\ y\ c_{\mathcal{B}^2} & = & fixB^2\ \mathcal{B}\ a\ x\ (fixB^2\ \mathcal{R}\ b\ y\ c) \\
fixB^2\ \mathcal{C}\ l\ k\ r & = & (\mathcal{C}, l, k, r)
\end{array}
\tag{4.13}
$$

如果没有匹配到上述三种模式，最后一行保持节点不变。双重黑色的修复是递归的。它中止于两种情况：一个是**情况 1**，双重黑色节点被消除了；另外一个是双重黑色向上移动，直到根节点，并最终恢复为黑色。下面的例子程序将以上情况汇总到一起：

```
-- the sibling is black, and has a red sub-tree
fixDB color a@(Node BB _ _ _) x (Node B (Node R b y c) z d)
      = Node color (Node B (shiftBlack a) x b) y (Node B c z d)
fixDB color BBEmpty x (Node B (Node R b y c) z d)
      = Node color (Node B Empty x b) y (Node B c z d)
fixDB color a@(Node BB _ _ _) x (Node B b y (Node R c z d))
      = Node color (Node B (shiftBlack a) x b) y (Node B c z d)
fixDB color BBEmpty x (Node B b y (Node R c z d))
      = Node color (Node B Empty x b) y (Node B c z d)
fixDB color (Node B a x (Node R b y c)) z d@(Node BB _ _ _)
      = Node color (Node B a x b) y (Node B c z (shiftBlack d))
fixDB color (Node B a x (Node R b y c)) z BBEmpty
      = Node color (Node B a x b) y (Node B c z Empty)
fixDB color (Node B (Node R a x b) y c) z d@(Node BB _ _ _)
      = Node color (Node B a x b) y (Node B c z (shiftBlack d))
fixDB color (Node B (Node R a x b) y c) z BBEmpty
      = Node color (Node B a x b) y (Node B c z Empty)
-- the sibling is red
fixDB B a@(Node BB _ _ _) x (Node R b y c)
      = fixDB B (fixDB R a x b) y c
fixDB B a@BBEmpty x (Node R b y c)
      = fixDB B (fixDB R a x b) y c
fixDB B (Node R a x b) y c@(Node BB _ _ _)
      = fixDB B a x (fixDB R b y c)
fixDB B (Node R a x b) y c@BBEmpty
      = fixDB B a x (fixDB R b y c)
-- the sibling and its 2 children are all black, move the blackness up
fixDB color a@(Node BB _ _ _) x (Node B b y c)
      = shiftBlack (Node color (shiftBlack a) x (Node R b y c))
fixDB color BBEmpty x (Node B b y c)
      = shiftBlack (Node color Empty x (Node R b y c))
fixDB color (Node B a x b) y c@(Node BB _ _ _)
      = shiftBlack (Node color (Node R a x b) y (shiftBlack c))
fixDB color (Node B a x b) y BBEmpty
      = shiftBlack (Node color (Node R a x b) y Empty)
-- otherwise
fixDB color l k r = Node color l k r
```

删除算法的复杂度为 $O(h)$，其中 h 为树的高度。由于红黑树保持平衡性，对于

n 个节点的树，$h = O(\lg n)$。

练习 4.5

1. 实现“标记——重建”删除算法：标记被删除的节点，但不进行真正的移除。当被标记的节点数目超过 50% 时重建树。

4.4 命令式红黑树算法 ⋆

通过模式匹配和递归，我们简化了红黑树的实现。为了完整，我们给出命令式的实现。插入算法的第一步和二叉搜索树相同，接下来通过旋转操作修复平衡。

```
1:  function INSERT(T, k)
2:      root ← T
3:      x ← CREATE-LEAF(k)
4:      COLOR(x) ← RED
5:      p ← NIL
6:      while T ≠ NIL do
7:          p ← T
8:          if k < KEY(T) then
9:              T ← LEFT(T)
10:         else
11:             T ← RIGHT(T)
12:     PARENT(x) ← p
13:     if p = NIL then                                  ▷ 树 T 为空
14:         return x
15:     else if k < KEY(p) then
16:         LEFT(p) ← x
17:     else
18:         RIGHT(p) ← x
19:     return INSERT-FIX(root, x)
```

新节点为红色，接下来修复平衡。共有 3 种基本情况，每种都有左右对称的情况，总计 6 种情况。其中有两种可以合并，它们都有红色的“叔父”节点，我们可将父节点和叔父节点都变为黑色，将祖父节点变为红色：

```
1:  function INSERT-FIX(T, x)
2:      while PARENT(x) ≠ NIL and COLOR(PARENT(x)) = RED do
3:          if COLOR(UNCLE(x)) = RED then        ▷ 情况 1：x 的叔父节点是红色
4:              COLOR(PARENT(x)) ← BLACK
5:              COLOR(GRAND-PARENT(x)) ← RED
```

```
6:          COLOR(UNCLE(x)) ← BLACK
7:          x ← GRAND-PARENT(x)
8:      else                                          ▷ x 的叔父节点是黑色
9:          if PARENT(x) = LEFT(GRAND-PARENT(x)) then
10:             if x = RIGHT(PARENT(x)) then          ▷ 情况 2：x 是右子树
11:                 x ← PARENT(x)
12:                 T ← LEFT-ROTATE(T, x)
                                                      ▷ 情况 3：x 是左子树
13:             COLOR(PARENT(x)) ← BLACK
14:             COLOR(GRAND-PARENT(x)) ← RED
15:             T ← RIGHT-ROTATE(T, GRAND-PARENT(x))
16:         else
17:             if x = LEFT(PARENT(x)) then           ▷ 情况 2 的对称
18:                 x ← PARENT(x)
19:                 T ← RIGHT-ROTATE(T, x)
                                                      ▷ 情况 3 的对称
20:             COLOR(PARENT(x)) ← BLACK
21:             COLOR(GRAND-PARENT(x)) ← RED
22:             T ← LEFT-ROTATE(T, GRAND-PARENT(x))
23:     COLOR(T) ← BLACK
24:     return T
```

插入算法的复杂度为 $O(\lg n)$，其中 n 是节点数。和 *balance* 函数对比，它们的处理逻辑并不相同。即使输入相同序列，也会构造出不同的红黑树。图4.11给出了两棵红黑树，它们是使用和图4.7中完全相同的序列构造出的。我们可以发现它们的不同。使用模式匹配的函数式算法存在一些性能损失。Okasaki 在 [13] 中给出了详细分析。

红黑树的命令式删除算法更为复杂，参见本书附录 A。

4.5 小结

红黑树是广泛使用的一种平衡二叉搜索树。我们在下一章介绍另外一种自平衡二叉树——AVL 树。红黑树可以看作是其它复杂的数据结构的基础：将子节点的数目扩展到 k 个，并且保持树的平衡，就可以演化到 B 树。如果将数据存储在边上，而非节点中，就演化出基数树。在红黑树的实现中，为了修复平衡性，需要处理多种情况。Okasaki 给出了一种简化方法，并激发了多种类似的实现 [16]。本书中的 AVL 树、Splay 树都是基于模式匹配方法实现的。

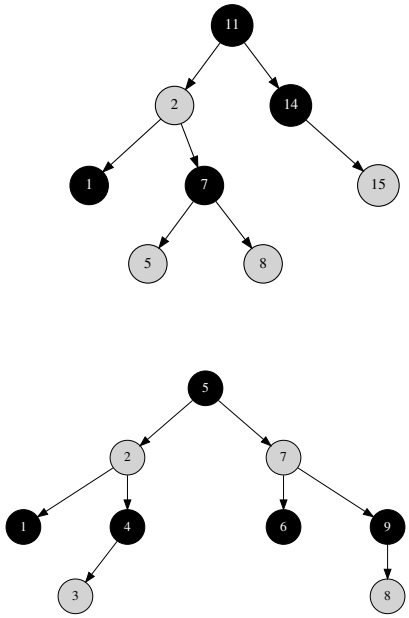


图 4.11: 命令式算法构建出的红黑树

4.6 附录：例子程序

带有父节点引用的红黑树定义，默认节点为红色。

```
data Node<T> {
    T key
    Color color
    Node<T> left
    Node<T> right
    Node<T> parent

    Node(T x) = Node(null, x, null, Color.RED)

    Node(Node<T> l, T k, Node<T> r, Color c) {
        left = l, key = k, right = r, color = c
        if left ≠ null then left.parent = this
        if right ≠ null then right.parent = this
    }

    Self setLeft(l) {
        left = l
        if l ≠ null then l.parent = this
    }

    Self setRight(r) {
        right = r
        if r ≠ null then r.parent = this
```

```
    }

    Node<T> sibling() = if parent.left == this then parent.right
                        else parent.left

    Node<T> uncle() = parent.sibling()

    Node<T> grandparent() = parent.parent
}
```

红黑树的插入：

```
Node<T> insert(Node<T> t, T key) {
    root = t
    x = Node(key)
    parent = null
    while (t ≠ null) {
        parent = t
        t = if (key < t.key) then t.left else t.right
    }
    if (parent == null) {    //tree is empty
        root = x
    } else if (key < parent.key) {
        parent.setLeft(x)
    } else {
        parent.setRight(x)
    }
    return insertFix(root, x)
}
```

插入后的平衡修复：

```
// Fix the red→red violation
Node<T> insertFix(Node<T> t, Node<T> x) {
    while (x.parent ≠ null and x.parent.color == Color.RED) {
        if (x.uncle().color == Color.RED) {
            // case 1: ((a:R x:R b) y:B c:R) ==> ((a:R x:B b) y:R c:B)
            x.parent.color = Color.BLACK
            x.grandparent().color = Color.RED
            x.uncle().color = Color.BLACK
            x = x.grandparent()
        } else {
            if (x.parent == x.grandparent().left) {
                if (x == x.parent.right) {
                    // case 2: ((a x:R b:R) y:B c) ==> case 3
                    x = x.parent
                    t = leftRotate(t, x)
                }
                // case 3: ((a:R x:R b) y:B c) ==> (a:R x:B (b y:R c))
                x.parent.color = Color.BLACK
                x.grandparent().color = Color.RED
                t = rightRotate(t, x.grandparent())
            } else {
                if (x == x.parent.left) {
```

```
                    // case 2': (a x:B (b:R y:R c)) ⟹ case 3'
                    x = x.parent
                    t = rightRotate(t, x)
                }
                // case 3': (a x:B (b y:R c:R)) ⟹ ((a x:R b) y:B c:R)
                x.parent.color = Color.BLACK
                x.grandparent().color = Color.RED
                t = leftRotate(t, x.grandparent())
            }
        }
    }
    t.color = Color.BLACK
    return t
}
```

第五章　AVL 树

本章介绍 AVL 树。同红黑树类似，AVL 树也是为了解决二叉树的平衡问题而提出的。它采用了更为直观的平衡性定义，以及恢复平衡的策略。对比红黑树和 AVL 树的设计和实现，有助于我们进一步理解自平衡二叉树的特性。

除了红黑树，还有没有其他自平衡二叉树呢？为了度量一棵二叉树的平衡，我们可以比较左右分支的高度差，如果差很大，则说明树不平衡。定义一棵树的高度差如下：

$$\delta(T) = |T_r| - |T_l| \tag{5.1}$$

其中 $|T|$ 代表树 T 的高度，T_l 和 T_r 分别代表左右分支。

若每棵子树都有 $\delta(T) = 0$，说明树是平衡的。例如，一棵高度为 h 的完全二叉树有 $n = 2^h - 1$ 个节点。除了叶子节点外，所有节点都含有两个非空的分支。完全二叉树的所有分支都满足 $\delta(T) = 0$。另外一个特殊的例子是空树：$\delta(\phi) = 0$。通常 $\delta(T)$ 的绝对值越小，说明树越平衡。

我们定义 $\delta(T)$ 为一棵二叉树的平衡因子。

5.1　AVL 树的定义

如果一棵二叉搜索树的所有子树都满足如下条件，我们称之为 AVL 树。

$$|\delta(T)| \leq 1 \tag{5.2}$$

AVL 树中所有子树平衡因子的绝对值都不大于 1，只可能是-1、0、1 这三个值。图5.1给出了一棵 AVL 树的例子。

为什么 AVL 树能保证平衡性呢？或者说为什么这个定义能保证一棵有 n 个节点的树的高度为 $O(\lg n)$？我们可以用下面的方法来证明这一事实。

对于一棵高为 h 的 AVL 树，它的节点数目并不是一个固定的值。当它是一棵完全二叉树时，含有的节点数目最多，为 $2^h - 1$。那么它最少包含多少节点呢？定义函数 $N(h)$ 代表高度为 h 的 AVL 树所含有的最少节点数目。对于简单的情况，我们可以立即得出 $N(h)$ 的值：

- 空树，$h = 0$，$N(0) = 0$；

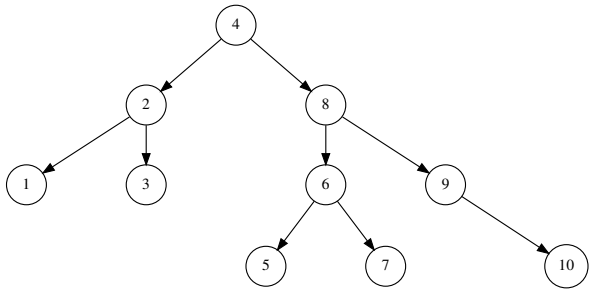


图 5.1: AVL 树的例子

- 只有一个根节点的树，$h = 1$，$N(1) = 1$；

一般情况下 $N(h)$ 是怎样的? 图5.2中给出了一个高度为 h 的 AVL 树 T。它包含三部分：根节点和左右两个分支 T_l、T_r。树的高度和子树高度之间满足下面的关系：

$$h = max(|T_l|, |T_r|) + 1 \tag{5.3}$$

因此，必然存在一个子树的高度为 $h-1$。根据 AVL 树的定义，我们有 $||T_l|-|T_r|| \leq 1$。所以另外一棵子树的高度不会小于 $h-2$。而 T 所包含的节点数为两个子树的节点数再加 1（1 个根节点）。于是我们得到下面的递归关系：

$$N(h) = N(h-1) + N(h-2) + 1 \tag{5.4}$$

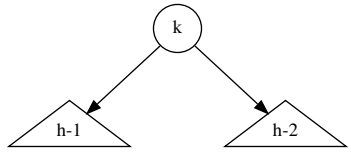


图 5.2: 高度为 h 的 AVL 树，其中一个分支高 $h-1$，另外一个分支的高度不小于 $h-2$

这一递归形式让我们联想起著名的斐波那契（Fibonacci）数列。如果定义 $N'(h) = N(h) + 1$，我们就可以将 (5.4) 转换成斐波那契数列。

$$N'(h) = N'(h-1) + N'(h-2) \tag{5.5}$$

引理 5.1.1. 若 $N(h)$ 表示高为 h 的 *AVL* 树的节点数目最小值, 令 $N'(h) = N(h)+1$, 则：

$$N'(h) \geq \phi^h \tag{5.6}$$

其中 $\phi = \frac{\sqrt{5}+1}{2}$，通常被称为黄金分割比。

证明. 使用数学归纳法。对于起始情况，我们有：

- $h = 0,\ N'(0) = 1 \geq \phi^0 = 1$
- $h = 1,\ N'(1) = 2 \geq \phi^1 = 1.618...$

对于递推情况，设 $N'(h) \geq \phi^h$。

$$\begin{aligned} N'(h+1) &= N'(h) + N'(h-1) && \{Fibonacci\} \\ &\geq \phi^h + \phi^{h-1} \\ &= \phi^{h-1}(\phi + 1) && \{\phi + 1 = \phi^2 = \frac{\sqrt{5}+3}{2}\} \\ &= \phi^{h+1} \end{aligned}$$

□

由定理5.1.1，我们立即得到下面的结果：

$$h \leq log_\phi(n+1) = log_\phi 2 \cdot \lg(n+1) \approx 1.44 \lg(n+1) \tag{5.7}$$

这一不等式说明 AVL 树的高度为 $O(\lg n)$，从而保证了平衡性。

在树的基本操作中，插入和删除会改变树的结构。如果由此导致平衡因子的绝对值超过 1，就需要通过修复使得 $|\delta|$ 恢复到 1 以内。常见的修复方法是使用树旋转。受到 Okasaki 在红黑树 [13] 中的思路启发，本章中，我们介绍一种模式匹配（pattern matching）方法。这种“改变 恢复”的操作，使得 AVL 树成为了一种自平衡二叉树。作为比较，本章同样也给出命令式的 AVL 树算法。

平衡因子 δ 显然可以通过递归求出。另外一种方法是在每个节点中保存一分平衡因子的值，如果树结构发生改变，我们只要更新这个值就可以了。这一方法不需要每次都进行遍历计算。

根据这一思路，我们在二叉搜索树的定义中增加一个 δ 变量，如下面的 C++ 代码所示[1]。

```
template <class T>
struct node {
    int delta;
    T key;
    node* left;
    node* right;
    node* parent;
};
```

某些纯函数式实现使用不同的构造函数（constructor）来保存平衡因子 δ。例如在 [17] 中，定义了 4 个 constructor：E、N、P、Z。其中，E 代表空树 ϕ；N 代表平衡因子为-1；P 代表平衡因子为 1；Z 代表平衡因子为 0。

本章中，我们直接在节点中保存平衡因子的值。

[1] 有些实现不保存 δ，取而代之保存树的高度，如 [20]。

```
data AVLTree a = Empty
               | Br (AVLTree a) a (AVLTree a) Int
```

我们将略过树的只读操作，包括查找、寻找最大、最小值等等，它们和二叉搜索树完全一样。我们仅关注哪些会改变树结构的操作。

5.2 插入

在 AVL 树中插入一个新元素可能会破坏平衡，使得平衡因子 δ 的绝对值超过 1。为了恢复平衡，可以根据不同的情形进行树的旋转操作。大多数命令式实现采用这种方法。

另外一种方法很像 Okasaki 在红黑树实现中使用的模式匹配方法。它的特点是简单直观。

向 AVL 树中插入一个新 key，根节点的平衡因子的变化 会在 $[-1, 1]$ 之间[2]，树的高度最多增加 1。我们需要递归地使用这一信息来更新其他层级上的平衡因子。定义插入算法的结果为一对值 $(T', \Delta H)$，其中 T' 为插入后的新树，ΔH 为树高度的增加值。令函数 $first(pair)$ 取得一对值中的第一个元素，我们可以在二叉搜索树的插入算法上进行改动，定义 AVL 树的插入操作：

$$insert(T, k) = first(ins(T, k)) \tag{5.8}$$

其中

$$ins(T, k) = \begin{cases} ((\phi, k, \phi, 0), 1) & : \quad T = \phi \\ tree(ins(T_l, k), k', (T_r, 0), \Delta) & : \quad k < k' \\ tree((T_l, 0), k', ins(T_r, k), \Delta) & : \quad otherwise \end{cases} \tag{5.9}$$

T_l、T_r、k'、Δ 的定义如下，它们分别表示左右子树，key 和平衡因子。

$$\begin{aligned} T_l &= left(T) \\ T_r &= right(T) \\ k' &= key(T) \\ \Delta &= \delta(T) \end{aligned}$$

向 AVL 树 T 中插入一个新 key k 时，如果树为空，结果为一个叶子节点，节点的 key 为 k，平衡因子为 0。树的高度增加 1。

否则，如果 T 不为空，我们需要比较根节点的 key k' 和待插入 key k 的大小。如果 k 小于根节点的 key，我们将其递归插入左子树，否则将其插入右子树。

根据定义，递归插入的结果为一对值，例如 $(T_l', \Delta H_l)$。我们需要对插入的结果调整平衡，并更新高度的增加值。为此定义函数 $tree()$，它接受 4 个参数：$(T_l', \Delta H_l)$、

[2]注意：这里不是说平衡因子的值在 $[-1, 1]$ 内，而是说它的变化在这个范围内。

k'、$(T'_r, \Delta H_r)$ 和 Δ。这一函数的运算结果记为 $(T', \Delta H)$。其中，T' 为调整平衡后的树，ΔH 是树高度的增加值，定义如下：

$$\Delta H = |T'| - |T| \tag{5.10}$$

它可以进一步分解为 4 种情况。

$$\begin{aligned}
\Delta H &= |T'| - |T| \\
&= 1 + max(|T'_r|, |T'_l|) - (1 + max(|T_r|, |T_l|)) \\
&= max(|T'_r|, |T'_l|) - max(|T_r|, |T_l|) \\
&= \begin{cases}
\Delta H_r & : \Delta \geq 0 \wedge \Delta' \geq 0 \\
\Delta + \Delta H_r & : \Delta \leq 0 \wedge \Delta' \geq 0 \\
\Delta H_l - \Delta & : \Delta \geq 0 \wedge \Delta' \leq 0 \\
\Delta H_l & : otherwise
\end{cases}
\end{aligned} \tag{5.11}$$

这一结论的详细证明可以参考附录 C。

在进行平衡调整前，我们还需要确定新的平衡因子 Δ'。根据 AVL 树平衡因子的定义，我们有：

$$\begin{aligned}
\Delta' &= |T'_r| - |T'_l| \\
&= |T_r| + \Delta H_r - (|T_l| + \Delta H_l) \\
&= |T_r| - |T_l| + \Delta H_r - \Delta H_l \\
&= \Delta + \Delta H_r - \Delta H_l
\end{aligned} \tag{5.12}$$

树高度的变化和平衡因子都准备好后，就可以定义 (5.9) 中的函数 $tree()$ 了。

$$tree((T'_l, \Delta H_l), k', (T'_r, \Delta H_r), \Delta) = balance((T'_l, k', T'_r, \Delta'), \Delta H) \tag{5.13}$$

在具体解释平衡调整的细节前，我们可以先给出上述函数的 Haskell 例子代码。首先是插入函数：

```
insert t x = fst $ ins t where
    ins Empty = (Br Empty x Empty 0, 1)
    ins (Br l k r d)
        | x < k     = tree (ins l) k (r, 0) d
        | x == k    = (Br l k r d, 0)
        | otherwise = tree (l, 0) k (ins r) d
```

这段代码中，如果待插入的 key 已经存在，它仅仅使用新 key 覆盖原先的值。

```
tree (l, dl) k (r, dr) d = balance (Br l k r d', delta) where
    d' = d + dr - dl
    delta = deltaH d d' dl dr
```

高度增加的计算函数定义如下：

```
deltaH d d' dl dr
       | d ≥0 && d' ≥0 = dr
       | d ≤0 && d' ≥0 = d+dr
       | d ≥0 && d' ≤0 = dl - d
       | otherwise = dl
```

5.2.1 平衡调整

我们准备使用模式匹配（pattern matching）来恢复平衡，首先需要考虑有哪些情况（pattern）会破坏 AVL 树的性质。

图5.3中展示了 4 种需要修复平衡的情况。这些情况中，平衡因子都是 2 或者-2，而不在范围 $[-1, 1]$ 之内。通过调整，平衡因子变成 0，左右分支变成同样的高度。

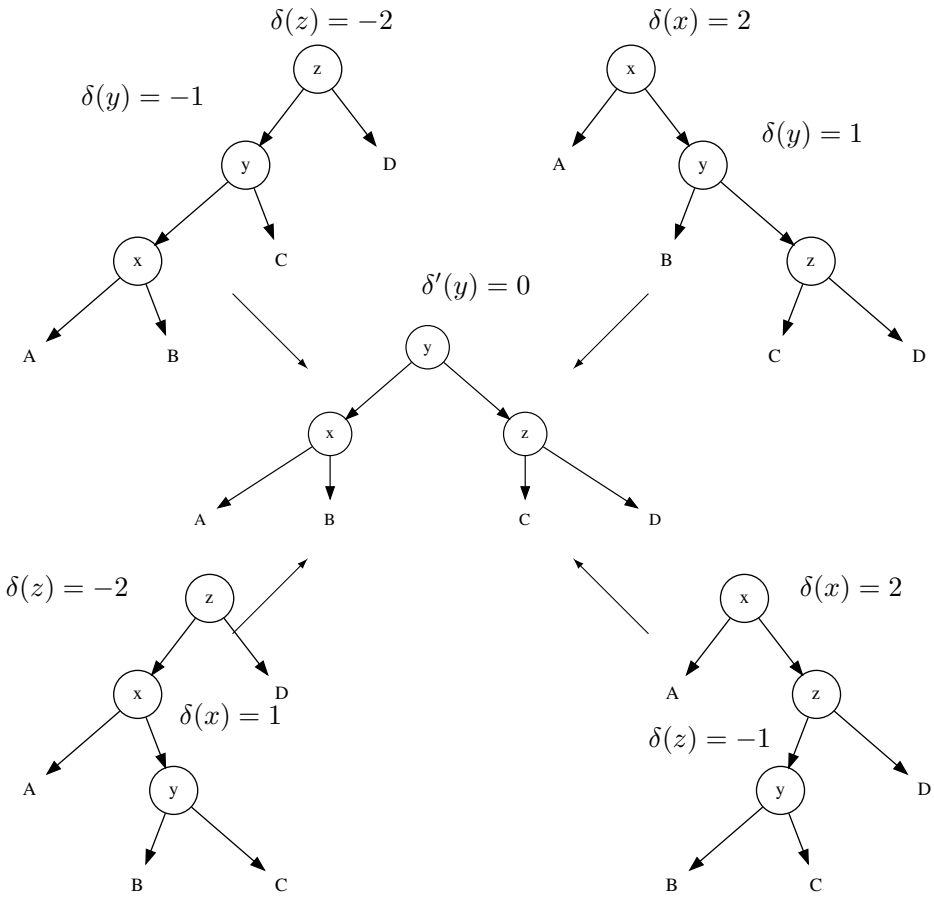


图 5.3: 插入后需要调整平衡的 4 种情况

我们从左上角开始，按照顺时针方向，依次称这 4 种情况为左－左偏（left-left lean）、右－右偏（right-right lean）、右－左偏（right-left lean）、左－右偏（left-right lean）。记调整前的平衡因子为 $\delta(x)$、$\delta(y)$、$\delta(z)$；调整后的平衡因子为 $\delta'(x)$、$\delta'(y)$、

$\delta'(z)$。

经过调整后，所有 4 种情况的平衡因子都变成 $\delta(y) = 0$。$\delta'(x)$ 和 $\delta'(z)$ 的结果如下。具体证明可以参考附录 C。

左－左偏（Left-left lean）的情况

$$\begin{aligned}\delta'(x) &= \delta(x) \\ \delta'(y) &= 0 \\ \delta'(z) &= 0\end{aligned} \tag{5.14}$$

右－右偏（Right-right lean）的情况

$$\begin{aligned}\delta'(x) &= 0 \\ \delta'(y) &= 0 \\ \delta'(z) &= \delta(z)\end{aligned} \tag{5.15}$$

右－左偏（Right-left lean）和左－右偏（Left-right lean）的情况

$$\begin{aligned}\delta'(x) &= \begin{cases} -1 & : \quad \delta(y) = 1 \\ 0 & : \quad otherwise \end{cases} \\ \delta'(y) &= 0 \\ \delta'(z) &= \begin{cases} 1 & : \quad \delta(y) = -1 \\ 0 & : \quad otherwise \end{cases}\end{aligned} \tag{5.16}$$

5.2.2 模式匹配

各种修复平衡的情况可以抽象成模式，下面的函数使用模式匹配定义了平衡修复算法。

$$balance(T, \Delta H) = \begin{cases} (((A, x, B, \delta(x)), y, (C, z, D, 0), 0), 0) & : \quad P_{ll}(T) \\ (((A, x, B, 0), y, (C, z, D, \delta(z)), 0), 0) & : \quad P_{rr}(T) \\ (((A, x, B, \delta'(x)), y, (C, z, D, \delta'(z)), 0), 0) & : \quad P_{rl}(T) \vee P_{lr}(T) \\ (T, \Delta H) & : \quad otherwise \end{cases} \tag{5.17}$$

其中 $P_{ll}(T)$ 表示树 T 满足左－左偏的情况。$\delta'(x)$ 和 $delta'(z)$ 按照式 (B.16) 定义。

$$\begin{aligned} P_{ll}(T) &: T = (((A, x, B, \delta(x)), y, C, -1), z, D, -2) \\ P_{rr}(T) &: T = (A, x, (B, y, (C, z, D, \delta(z)), 1), 2) \\ P_{rl}(T) &: T = ((A, x, (B, y, C, \delta(y)), 1), z, D, -2) \\ P_{lr}(T) &: T = (A, x, ((B, y, C, \delta(y)), z, D, -1), 2) \end{aligned} \tag{5.18}$$

下面的 Haskell 例子代码实现了这一平衡修复函数。

```
balance (Br (Br (Br a x b dx) y c (-1)) z d (-2), _) =
        (Br (Br a x b dx) y (Br c z d 0) 0, 0)
balance (Br a x (Br b y (Br c z d dz)    1)    2, _) =
        (Br (Br a x b 0) y (Br c z d dz) 0, 0)
balance (Br (Br a x (Br b y c dy)    1) z d (-2), _) =
        (Br (Br a x b dx') y (Br c z d dz') 0, 0) where
    dx' = if dy ==  1 then -1 else 0
    dz' = if dy == -1 then  1 else 0
balance (Br a x (Br (Br b y c dy) z d (-1))    2, _) =
        (Br (Br a x b dx') y (Br c z d dz') 0, 0) where
    dx' = if dy ==  1 then -1 else 0
    dz' = if dy == -1 then  1 else 0
balance (t, d) = (t, d)
```

插入算法的性能和树的高度成正比，根据之前给出的证明，如果 AVL 树包含 n 个元素，插入算法的性能为 $O(\lg n)$。

验证

可以定义一个函数来检查一棵树是否是 AVL 树。我们需要验证两方面：首先它必须是一棵合法的二叉搜索树；其次它满足 AVL 树的性质。

我们略过二叉搜索树的检验，把它留给读者作为练习。

为了验证 AVL 树的性质是否满足，我们需要检查左右分支的高度差，然后再递归检查左右分支是否也满足 AVL 树的性质。直到最终到达叶子节点。

$$avl?(T) = \begin{cases} True & : \quad T = \phi \\ avl?(T_l) \wedge avl?(T_r) \wedge ||T_r| - |T_l|| \leq 1 & : \quad otherwise \end{cases} \tag{5.19}$$

树的高度可以根据定义递归进行计算：

$$|T| = \begin{cases} 0 & : \quad T = \phi \\ 1 + max(|T_r|, |T_l|) & : \quad otherwise \end{cases} \tag{5.20}$$

相应的 Haskell 例子程序实现如下：

```
isAVL Empty = True
isAVL (Br l _ r d) = and [isAVL l, isAVL r, abs (height r - height l) ≤ 1]

height Empty = 0
height (Br l _ r _) = 1 + max (height l) (height r)
```

练习 5.1

编写程序检查一棵二叉树是否是二叉搜索树。如果使用命令式（imperative）语言，请考虑如何消除递归。

5.3 删除

我们在二叉搜索树的章节曾经解释过，在纯函数式的环境中删除操作意义不大。由于树是只读的，它通常是在一次性构建之后用于反复查询。

我们曾经在红黑树一章中实现了删除，它本质上是重新构建一棵新树。AVL 树的删除算法可以参考附录 C。

5.4 AVL 树的命令式算法 ⋆

我们已经介绍了 AVL 树相关的主要内容。本节我们展示传统的 AVL 树“插入—旋转”算法，读者可以将其和模式匹配算法进行比较。

和红黑树的命令式插入算法相似，我们先按照普通二叉搜索树将新元素插入，然后再通过旋转操作恢复平衡。

```
1: function INSERT(T, k)
2:     root ← T
3:     x ← CREATE-LEAF(k)
4:     δ(x) ← 0
5:     parent ← NIL
6:     while T ≠ NIL do
7:         parent ← T
8:         if k < KEY(T) then
9:             T ← LEFT(T)
10:        else
11:            T ← RIGHT(T)
12:    PARENT(x) ← parent
13:    if parent = NIL then                          ▷ 树 T 为空
14:        return x
15:    else if k < KEY(parent) then
16:        LEFT(parent) ← x
17:    else
18:        RIGHT(parent) ← x
19:    return AVL-INSERT-FIX(root, x)
```

插入新元素后，树的高度可能增加，因此平衡因子 δ 也会变化。插入到右侧会使 δ 增加 1，插入左侧会使 δ 减少 1。在算法结束前，我们需要从 x 开始，自底向上修复平衡，直到根节点。

下面的 Python 例子程序实现了插入算法的主要部分。

```
def avl_insert(t, key):
```

```
    root = t
    x = Node(key)
    parent = None
    while(t):
        parent = t
        if(key < t.key):
            t = t.left
        else:
            t = t.right
    if parent is None: #tree is empty
        root = x
    elif key < parent.key:
        parent.set_left(x)
    else:
        parent.set_right(x)
    return avl_insert_fix(root, x)
```

算法首先自顶向下从根开始搜索插入位置，然后将新元素作为叶子节点插入。最后它调用修复程序，并传入根和新插入的节点。

这里我们复用了红黑树一章中定义的 `set_left()` 和 `set_right()` 方法。

为了修复平衡，我们需要检查新节点是插入到了左侧还是右侧。如果在左侧，平衡因子 δ 减小，否则增加。记新的平衡因子为 δ'，我们有如下三种情况：

- 若 $|\delta| = 1$ 而 $|\delta'| = 0$，说明插入后树处于平衡状态。父节点的高度没有发生变化，算法结束。
- 若 $|\delta| = 0$ 而 $|\delta'| = 1$，说明左右分支之一的高度增加了，我们需要继续向上检查树的平衡性。
- 若 $|\delta| = 1$ and $|\delta'| = 2$，说明 AVL 树不再平衡了，我们需要进行旋转操作进行修复。

```
1:  function AVL-Insert-Fix(T, x)
2:      while Parent(x) ≠ NIL do
3:          δ ← δ(Parent(x))
4:          if x = Left(Parent(x)) then
5:              δ' ← δ − 1
6:          else
7:              δ' ← δ + 1
8:          δ(Parent(x)) ← δ'
9:          P ← Parent(x)
10:         L ← Left(x)
11:         R ← Right(x)
12:         if |δ| = 1 and |δ'| = 0 then          ▷ 高度没有变化，结束。
13:             return T
```

```
14:     else if |δ| = 0 and |δ'| = 1 then                ▷ 继续自底向上进行更新。
15:         x ← P
16:     else if |δ| = 1 and |δ'| = 2 then
17:         if δ' = 2 then
18:             if δ(R) = 1 then                          ▷ 右-右情况
19:                 δ(P) ← 0                              ▷ 根据式 (B.5)
20:                 δ(R) ← 0
21:                 T ← LEFT-ROTATE(T, P)
22:             if δ(R) = −1 then                         ▷ 右-左情况
23:                 δ_y ← δ(LEFT(R))                      ▷ 根据式 (B.16)
24:                 if δ_y = 1 then
25:                     δ(P) ← −1
26:                 else
27:                     δ(P) ← 0
28:                 δ(LEFT(R)) ← 0
29:                 if δ_y = −1 then
30:                     δ(R) ← 1
31:                 else
32:                     δ(R) ← 0
33:                 T ← RIGHT-ROTATE(T, R)
34:                 T ← LEFT-ROTATE(T, P)
35:         if δ' = −2 then
36:             if δ(L) = −1 then                         ▷ 左-左情况
37:                 δ(P) ← 0
38:                 δ(L) ← 0
39:                 RIGHT-ROTATE(T, P)
40:             else                                      ▷ 左-右情况
41:                 δ_y ← δ(RIGHT(L))
42:                 if δ_y = 1 then
43:                     δ(L) ← −1
44:                 else
45:                     δ(L) ← 0
46:                 δ(RIGHT(L)) ← 0
47:                 if δ_y = −1 then
48:                     δ(P) ← 1
49:                 else
50:                     δ(P) ← 0
```

51: LEFT-ROTATE(T, L)
52: RIGHT-ROTATE(T, P)
53: break
54: **return** T

这里我们复用了红黑树一章中定义的旋转操作。单纯旋转操作并不更新平衡因子 δ。由于旋转变换改变了树结构，增加了平衡性，因此我们需要重新计算平衡因子。这里直接使用了上面的结果。在 4 种情况中，右－右偏和左－左偏需要进行一次旋转；而右－左偏和左－右偏需要进行两次旋转。

相关的 Python 例子程序如下：

```
def avl_insert_fix(t, x):
    while x.parent is not None:
        d2 = d1 = x.parent.delta
        if x == x.parent.left:
            d2 = d2 - 1
        else:
            d2 = d2 + 1
        x.parent.delta = d2
        (p, l, r) = (x.parent, x.parent.left, x.parent.right)
        if abs(d1) == 1 and abs(d2) == 0:
            return t
        elif abs(d1) == 0 and abs(d2) == 1:
            x = x.parent
        elif abs(d1)==1 and abs(d2) == 2:
            if d2 == 2:
                if r.delta == 1: #右-右情况
                    p.delta = 0
                    r.delta = 0
                    t = left_rotate(t, p)
                if r.delta == -1: #右-左情况
                    dy = r.left.delta
                    if dy == 1:
                        p.delta = -1
                    else:
                        p.delta = 0
                    r.left.delta = 0
                    if dy == -1:
                        r.delta = 1
                    else:
                        r.delta = 0
                    t = right_rotate(t, r)
                    t = left_rotate(t, p)
            if d2 == -2:
                if l.delta == -1: #左-左情况
                    p.delta = 0
                    l.delta = 0
                    t = right_rotate(t, p)
                if l.delta == 1: #左-右情况
                    dy = l.right.delta
                    if dy == 1:
```

```
                    l.delta = -1
                else:
                    l.delta = 0
                l.right.delta = 0
                if dy == -1:
                    p.delta = 1
                else:
                    p.delta = 0
                t = left_rotate(t, l)
                t = right_rotate(t, p)
            break
    return t
```

命令式的 AVL 书删除算法可以参考附录 C。

5.5 小结

AVL 树是在 1962 年由 Adelson-Velskii 和 Landis[18]、[19] 发表的。AVL 树的命名来自两位作者的名字。它的历史要比红黑树更早。

人们很自然会比较 AVL 树和红黑树。它们都是自平衡二叉搜索树，对于主要的树操作，它们的性能都是 $O(\lg n)$ 的。根据式 (5.7) 的结果，AVL 树的平衡性更为严格，因此在频繁查询的情况下，其表现要好于红黑树 [18]。但红黑树在频繁插入和删除的情况下性能更佳。

很多流行的程序库使用红黑树作为自平衡二叉搜索树的内部实现，例如 STL，AVL 树同样也可以直观、高效地解决平衡问题。

在接下来的章节里，我们将介绍一些数据结构，它们使用边，而不是节点来存储信息，如 Trie 和 Patricia 等等。在保证平衡的情况下，如果允许含有两个以上的子分支，我们就得到了另一种有趣的数据结构— B 树。

第六章　基数树— Trie 和前缀树

6.1　简介

前面章节介绍的各种树都是利用节点来存储信息，我们也可以利用边（edge）来存储。基数树（Radix tree），如 Trie 和前缀树就是这样的数据结构。它们产生于 1960 年代，被广泛用于编译器 [21] 和生物信息处理（如 DNA 模式匹配）[23] 等领域。

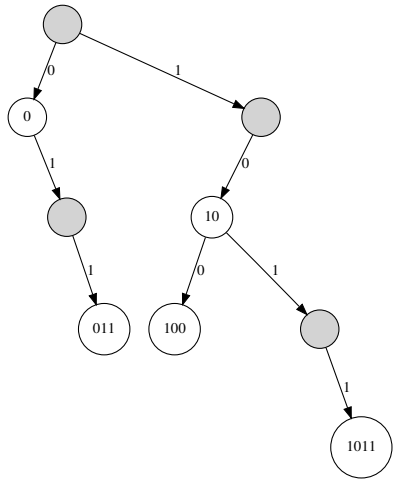


图 6.1: 基数树

图6.1展示了一棵基数树（[4] 第 269 页）。它包含了串 1011、10、011、100 和 0。如果要在其中查找二进制数 $k = (b_0 b_1 ... b_n)_2$，我们首先检查左侧的最高位 b_0 是 1 还是 0，若为 0，则接下来在左侧分支查找；若为 1，则继续在右侧分支查找。接着，我们检查第二位，并重复这一过程直到处理完所有的 n 位或到达某一叶子节点。

基数树并不在节点中存储 key，信息由边来代表。图中节点中标注的 key 仅仅是为了方便理解。使用整数的二进制串来代表 key，既可以节省空间，又可以通过位运算，加快处理速度。

6.2 整数 Trie

图6.1中所示的数据结构常被称为 *binary trie*。Trie 是 Edward Fredkin 提出的。它来自英文单词 retrieval，最开始发音为/'tri:/。但是许多人都把它读作/'trai/(和英文单词 try 的发音相同)[24]。Trie 也被称为前缀树。一棵 binary trie 是一种特殊的二叉树，每个 key 存放的位置由它的二进制位来决定。0 表示“向左”，而 1 表示“向右”[21]。

由于整数可以表示为二进制，我们可以用整数代替 0/1 串作为 key。当把一个整数插入到 Trie 时，我们首先将其转化为二进制，然后检查第一位，若为 0，则递归插入到左侧分支，若为 1，则插入到右侧分支。但是这里有一个问题。考虑图6.2中的 Trie，如果使用 0/1 串，这 3 个 key 各不相同，但它们却代表同一个十进制整数。对于这个例子来说，我们应该在 Trie 中的哪个位置插入整数 3 呢?

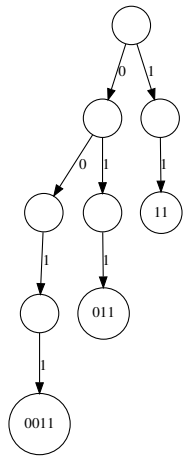


图 6.2: 大端（big-endian）Trie

如果把前面的 0 也当作有效位，在一个 32 位整数的系统中，向空 Trie 插入 1，结果将是一棵有 32 层分支的树。其中的 31 个中间节点只有一个左子树，最后一个节点只有一个右子树。空间利用率很低。

Okasaki 给出了一种解决方法 [21]。二进制数的最高位（MSB）通常在左边，最低位（LSB）在右边。这种 Trie 称作大端（big-endian）树。我们也可以使用小端（little-endian）数来表示 key。这样，十进制的 1 表示为小端二进制的 1。如果插入到空 Trie 中，结果就是含有一个右侧叶子节点的树。它只有一层分支。十进制的 2 将被表示为小端二进制的 01，十进制的 3 表示为小端二进制 $(11)_2$。这样就消除了有效位前面的 0，每一个整数 key 在 Trie 中的位置可以被唯一确定。

6.2.1 整数 Trie 的定义

我们可以重用二叉树的结构来定义 binary trie。一个 binary trie 的节点要么为空，要么包含左右两个分支。非空节点可以保存额外的附加数据（satellite data）。左侧分支编码为 0，右侧分支编码为 1。

下面的 Haskell 例子代码定义了 Trie 的代数数据类型。

```
data IntTrie a = Empty
               | Branch (IntTrie a) (Maybe a) (IntTrie a)
```

在命令式编程语言中，Trie 通常被定义为结构或类，如下面的 Python 例子代码所示：

```
class IntTrie:
    def __init__(self):
        self.left = self.right = None
        self.value = None
```

6.2.2 插入

由于整数 Trie 的定义是递归的，插入算法可以很自然地用递归进行定义。如果最左侧的位为 0，说明待插入的 key 为偶数，我们接下来在左侧分支递归插入；否则最左侧的位为 1，说明 key 为奇数，我们转向右侧分支。我们这样不断地将 key 除以 2 取整以去掉最左侧的位，直到处理完所有的位。此时 key 变为 0，插入结束。记 Trie 树 T 的左右分支为 T_l 和 T_r，节点上存储的数据为 d（可以为空）。如果 T 为空，则其左右分支和数据也定义为空。我们可以这样定义插入算法：

$$insert(T, k, v) = \begin{cases} (T_l, v, T_r) & : & k = 0 \\ (insert(T_l, k/2, v), d, T_r) & : & even(k) \\ (T_l, d, insert(T_r, \lfloor k/2 \rfloor, v)) & : & otherwise \end{cases} \tag{6.1}$$

如果待插入的 key 已经存在，这一算法会覆盖原来存储的数据。也可以使用链表保存同一 key 上的所有数据。

图6.2的例子是依次插入射对 $\{1 \to a, 4 \to b, 5 \to c, 9 \to d\}$ 的结果。

下面的 Haskell 例子程序实现了这一插入算法。

```
insert t 0 x = Branch (left t) (Just x) (right t)
insert t k x
    | even k = Branch (insert (left t) (k `div` 2) x) (value t) (right t)
    | otherwise = Branch (left t) (value t) (insert (right t) (k `div` 2) x)

left (Branch l _ _) = l
left Empty = Empty

right (Branch _ _ r) = r
right Empty = Empty
```

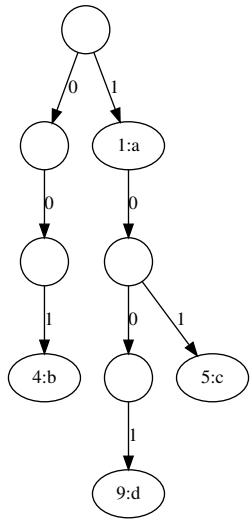


图 6.3: 用小端整数 binary trie 实现的映射：$\{1 \to a, 4 \to b, 5 \to c, 9 \to d\}$

```
value (Branch _ v _) = v
value Empty = Nothing
```

也可以用命令式的方式定义插入算法。由于 key 是小端整数，插入时，我们需要从右侧逐位进行处理。若为 0，则递归插入左子树；若为 1，则插入右子树。如果子树为空，我们需要创建一个新节点。重复这一步骤直到处理完最后一位（最左侧的位）后停止。

```
1: function INSERT(T, k, v)
2:     if T = NIL then
3:         T ← EMPTY-NODE
4:     p ← T
5:     while k ≠ 0 do
6:         if EVEN?(k) then
7:             if LEFT(p) = NIL then
8:                 LEFT(p) ← EMPTY-NODE
9:             p ← LEFT(p)
10:        else
11:            if RIGHT(p) = NIL then
12:                RIGHT(p) ← EMPTY-NODE
13:            p ← RIGHT(p)
14:        k ← ⌊k/2⌋
15:    DATA(p) ← v
16:    return T
```

插入算法接受 3 个参数：一棵 Trie 树 T、要插入的整数 k 和相应的数据 v。下面的 Python 例子程序实现了这一算法。这段例子程序使用了位运算来判断奇偶和移位。

```
def insert(t, key, value = None):
    if t is None:
        t = IntTrie()
    p = t
    while key ≠ 0:
        if key & 1 == 0:
            if p.left is None:
                p.left = IntTrie()
            p = p.left
        else:
            if p.right is None:
                p.right = IntTrie()
            p = p.right
        key = key >> 1  # key / 2
    p.value = value
    return t
```

对于有 m 位的二进制整数 k，这一算法递归 m 次，因此时间复杂度为 $O(m)$。

6.2.3 查找

在小端整数 trie 中查找数字 k 时，若树为空，则显然 k 不存在；如果 $k = 0$，则返回当前根节点中存储的数据。否则根据最后一位是 0 还是 1，在左右分支进行递归查找。

$$lookup(T, k) = \begin{cases} \phi & : & T = \phi \\ d & : & k = 0 \\ lookup(T_l, k/2) & : & even(k) \\ lookup(T_r, \lfloor k/2 \rfloor) & : & otherwise \end{cases} \tag{6.2}$$

下面的 Haskell 例子程序实现了递归查找算法。

```
search Empty k = Nothing
search t 0 = value t
search t k = if even k then search (left t) (k `div` 2)
             else search (right t) (k `div` 2)
```

也可以用命令式的方式实现查找，我们从 k 的二进制形式最右侧开始，逐位检查，如果为 0，就继续在左侧分支查找；如果为 1，则在右侧分支查找。直到所有位都处理完毕。

1: **function** LOOKUP(T, k)
2: **while** $k \neq 0 \wedge T \neq$NIL **do**
3: **if** EVEN?(k) **then**
4: $T \leftarrow$ LEFT(T)

```
5:          else
6:              T ← RIGHT(T)
7:          k ← ⌊k/2⌋
8:      if T ≠ NIL then
9:          return DATA(T)
10:     else
11:         return not found
```

下面的 Python 例子程序实现了查找算法。

```
def lookup(t, key):
    while t is not None and k ≠ 0:
        if key & 1 == 0:
            t = t.left
        else:
            t = t.right
        key = key >> 1
    return None if t is None else t.value
```

若待查找的 key 有 m 位，则查找算法的复杂度为 $O(m)$。

6.3 整数前缀树

Trie 的缺点是空间消耗大。如图6.2所示，只有叶子节点存储了最终的数据。整数 Trie 中有大量的节点只包含一棵子树。为了提高空间利用率，我们可以将一连串“独生子女”压缩成一个节点。前缀树就是这样的数据结构，由 Donald R. Morrison 在 1968 年提出。在他的论文中，前缀树被称为 Patricia。是 **P**ractical **A**lgorithm **T**o **R**etrieve **I**nformation **C**oded **I**n **A**lphanumeric 的首字母缩写 [22]。

Okasaki 给出了整数前缀树的实现 [21]。将图6.3中只有一个子树的节点合并后，可以得到一棵如图6.4所示的树。

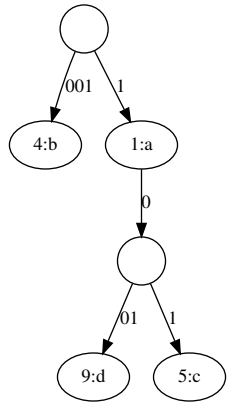


图 6.4: 小端前缀树实现的映射 $\{1 \to a, 4 \to b, 5 \to c, 9 \to d\}$

观察此图，可以发现分支节点所代表的 key 是它的所有子分支的公共前缀。这些子分支 key 的开头部分一样，然后从某一点开始出现不同。和 Trie 相比，前缀树节省了很多空间。

和整数 Trie 不同，整数前缀树可以使用大端实现而不会遇到6.2中所描述的 0 前缀问题。第一位有效数字前的 0 全都被去除以节省空间。Okasaki 在 [21] 中列出了大端前缀树的优点。

6.3.1 定义

整数前缀树是一种特殊的二叉树。它或者为空，或者是一个节点。节点有两种类型：

- 叶子节点：包含一个整数 key 和相应的数据（数据可为空）；
- 分支节点：包含左右子分支。两个子分支的 key 具有最长的二进制公共前缀。其中左侧子分支的下一位是 0，而右侧子分支的下一位是 1。

下面的 Haskell 例子代码定义了前缀树：

```
type Key = Int
type Prefix = Int
type Mask = Int

data IntTree a = Empty
               | Leaf Key a
               | Branch Prefix Mask (IntTree a) (IntTree a)
```

为了表示从哪一位开始左右分支的 key 变得不相同，分支节点中保存了掩码信息。通常掩码是 2 的整数次幂，形如 2^n，其中 n 是非负整数。所有低于 n 的二进制位都不属于 key 的公共前缀。

下面的 Python 例子代码定义了前缀树和相应的辅助函数。

```
class IntTree:
    def __init__(self, key = 0, value = None):
        self.key = key
        self.value = value
        self.prefix = key
        self.mask = 1
        self.left = self.right = None

    def isleaf(self):
        return self.left is None and self.right is None

    def replace(self, x, y):
        if self.left == x:
            self.left = y
        else:
            self.right = y
```

```
    def match(self, k):
        return maskbit(k, self.mask) == self.prefix
```

其中 match 判断节点的前缀在掩码前是否和给定的 key 一致。我们稍后介绍它的实现。

6.3.2 插入

当插入 key 时，如果树为空，结果为一个叶子节点，key 和相关的数据存储于节点中，如图6.5所示。

图 6.5: 左侧：树为空；右侧：插入整数 12 后

如果树只有一个叶子节点 x，我们把待插入的 key 和数据放入一个新的叶子节点 y 中。然后创建一个新的分支节点，并令 x 和 y 为这一新分支节点的两个子节点。为了确定 y 应该在左边还是右边，我们需要找到 x 和 y 的最长公共前缀。举个例子，假设 $key(x)$ 为 12（二进制 1100）,$key(y)$ 为 15（二进制 1111），则最长公共前缀为二进制 $11oo$，其中 o 代表我们不关心的二进制位，我们可以使用一个整数通过掩码来去掉这些位。在这个例子中，可以用 4（二进制 100）作为掩码。最长公共前缀后面的一位代表 2^1。$key(x)$ 中这一位是 0，而 $key(y)$ 中这一位是 1。因此 x 是左子树，而 y 是右子树。这个例子如图6.6所示。

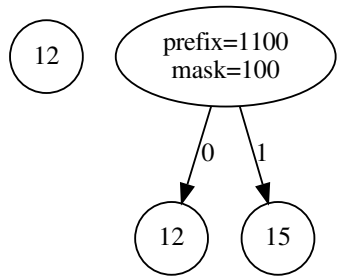
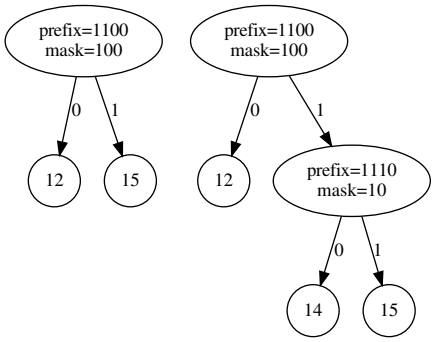


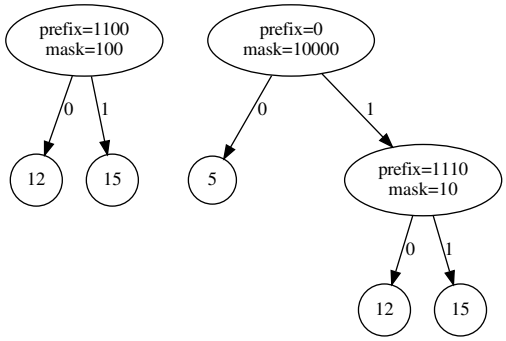
图 6.6: 左侧：只含有一个叶子节点 12 的树；右侧：插入整数 15 后

如果树既不为空，也不是一个单独的叶子节点，我们需要先比较待插入的 key 和根节点中记录的最长公共前缀是否一致。如果一致，则根据接下来的位是 0 还是 1 递归地在左侧或右侧进行插入。例如，若将整数 14（二进制 1110）插入图6.6中所示的树中，由于最长公共前缀是 $11oo$ 而接下来的一位（2^1 位）是 1，所以需要将 14 递归插入到右子树。

最后，如果待插入的整数和根节点中记录的最长公共前缀不一致，我们需要从根节点分出一棵新的枝杈。图6.7展示了这两种不同的情况。



(a) 插入整数 14。它和最长公共前缀 $(1100)_2$ 一致。需要将其递归插入到右侧分支中。



(b) 插入整数 5。它和最长公共前缀 $(1100)_2$ 不一致。需要新分杈出一个分支。

图 6.7: 向分支节点插入整数

记要插入的整数为 k，数据为 v 的节点为 (k, v)，分支节点记为 (p, m, T_l, T_r)，其中 p 代表最长公共前缀，m 表示掩码，T_l 和 T_r 分别代表左右子分支。上述情况可以归纳为下面的插入算法：

$$insert(T, k, v) = \begin{cases} (k, v) & : & T = \phi \vee T = (k, v') \\ join(k, (k, v), k', T) & : & T = (k', v') \\ (p, m, insert(T_l, k, v), T_r) & : & T = (p, m, T_l, T_r), match(k, p, m), zero(k, m) \\ (p, m, T_l, insert(T_r, k, v)) & : & T = (p, m, T_l, T_r), match(k, p, m), \neg zero(k, m) \\ join(k, (k, v), p, T) & : & T = (p, m, T_l, T_r), \neg match(k, p, m) \end{cases} \tag{6.3}$$

第一行处理边界情况，T 或者为空，或者是一个具有同样 key 的叶子节点。我们用新的值覆盖此前的数据。

第二行处理 T 为叶子节点，但是 key 不同的情况。这时需要分支出一个新的叶子节点，为此，我们需要计算出最长公共前缀，并判断哪个在左侧，哪个在右侧。函数 $join(k_1, T_1, k_2, T_2)$ 负责这些处理，我们稍后会定义它。

第三、四行处理 T 为分支节点，且分支代表的前缀和待插入的整数一致的情况。如果接下来的一位是 0，则第三行会递归地向左侧分支进行插入。否则第四行递归地向右侧分支插入。

最后一行处理 T 为分支节点，但是 key 不一致的情况。我门需要调用 $join$ 函数来分支出一个新的叶子节点。

接下来需要定义函数 $match(k, p, m)$ 用以判断整数 k 在掩码 m 以上的位是否和 p 一致。也就是检查 p 在掩码以上的位是否为 k 的一个前缀。例如，一个分支节点的 key 表示为二进制 $(p_n p_{n-1} ... p_i ... p_0)_2$，待插入的 key k 的二进制形式为 $(k_n k_{n-1} ... k_i ... k_0)_2$，掩码为 $(100...0)_2 = 2^i$。则称 k、p、m 一致当且仅当对于任意 $j, i \leq j \leq n$ 有 $p_j = k_j$。

我们可以通过判断等式 $mask(k, m) = p$ 是否成立来实现 $match$ 函数。其中 $mask(x, m) = \overline{m-1} \& x$。即先对 $m-1$ 按位取反，然后将结果和 x 按位进行与运算。

函数 $zero(k, m)$ 检查公共前缀接下来的一位是否为 0。我们可以将掩码 m 向右做 1 位移位运算，接下来和 k 进行按位与运算。

$$zero(k, m) = k \& shift_r(m, 1) \tag{6.4}$$

举例来说，若 $m = (100...0)_2 = 2^i$、$k = (k_n k_{n-1} ... k_i 1 ... k_0)_2$，由于 k_i 的下一位是 1，所以 $zero(k, m)$ 的结果为 false；反之，若 $k = (k_n k_{n-1} ... k_i 0 ... k_0)_2$，则结果为 true。

函数 $join(p_1, T_1, p_2, T_2)$ 接受两个前缀和两棵树作为参数。它找出 p_1 与 p_2 的最长公共前缀，然后创建一个新的分支节点，并将 T_1 和 T_2 作为子节点。

$$join(p_1, T_1, p_2, T_2) = \begin{cases} (p, m, T_1, T_2) & : & zero(p1, m), (p, m) = LCP(p_1, p_2) \\ (p, m, T_2, T_1) & : & \neg zero(p1, m) \end{cases} \tag{6.5}$$

为了计算 p_1 和 p_2 的最长公共前缀，我们可以先对它们计算异或，然后用这个结

果的有效位数产生一个掩码 $m = 2^{|xor(p_1,p_2)|}$。最长公共前缀就可以用这个掩码和 p_1 与 p_2 中的任何一个得出。例如：

$$p = mask(p_1, m) \tag{6.6}$$

下面的 Haskell 例子程序实现了插入算法：

```
import Data.Bits

insert t k x
  = case t of
      Empty → Leaf k x
      Leaf k' x' → if k==k' then Leaf k x
                   else join k (Leaf k x) k' t -- t@(Leaf k' x')
      Branch p m l r
        | match k p m → if zero k m
                        then Branch p m (insert l k x) r
                        else Branch p m l (insert r k x)
        | otherwise → join k (Leaf k x) p t -- t@(Branch p m l r)

join p1 t1 p2 t2 = if zero p1 m then Branch p m t1 t2
                                else Branch p m t2 t1
    where
      (p, m) = lcp p1 p2

lcp :: Prefix → Prefix → (Prefix, Mask)
lcp p1 p2 = (p, m) where
    m = bit (highestBit (p1 `xor` p2))
    p = mask p1 m

highestBit x = if x == 0 then 0 else 1 + highestBit (shiftR x 1)

mask x m = (x .&. complement (m-1)) -- complement 表示按位取反

zero x m = x .&. (shiftR m 1) == 0

match k p m = (mask k m) == p
```

插入算法也可以用命令式方法实现：

```
1:  function INSERT(T, k, v)
2:      if T = NIL then
3:          T ← CREATE-LEAF(k, v)
4:          return T
5:      y ← T
6:      p ← NIL
7:      while y is not leaf, and MATCH(k, PREFIX(y), MASK(y)) do
8:          p ← y
9:          if ZERO?(k, MASK(y)) then
10:             y ← LEFT(y)
```

```
        else
            y ← RIGHT(y)
    if y is leaf, and k = KEY(y) then
        DATA(y) ← v
    else
        z ← BRANCH(y, CREATE-LEAF(k, v))
        if p = NIL then
            T ← z
        else
            if LEFT(p) = y then
                LEFT(p) ← z
            else
                RIGHT(p) ← z
    return T
```

函数 $\textsc{Branch}(T_1, T_2)$ 的作用和前面定义的 $join$ 类似。它创建一个新的分支节点，计算最长公共前缀，然后将 T_1 和 T_2 设置为两棵子树。

```
function BRANCH(T_1, T_2)
    T ← EMPTY-NODE
    ( PREFIX(T), MASK(T) ) ← LCP(PREFIX(T_1), PREFIX(T_2))
    if ZERO?(PREFIX(T_1), MASK(T)) then
        LEFT(T) ← T_1
        RIGHT(T) ← T_2
    else
        LEFT(T) ← T_2
        RIGHT(T) ← T_1
    return T
```

下面的 Python 例子程序实现了插入算法:

```
def insert(t, key, value):
    if t is None:
        return IntTree(key, value)
    node = t
    parent = None
    while (not node.isleaf()) and node.match(key):
        parent = node
        if zero(key, node.mask):
            node = node.left
        else:
            node = node.right
    if node.isleaf() and key == node.key:
        node.value = value
    else:
```

```
        p = branch(node, IntTree(key, value))
        if parent is None:
            return p
        parent.replace(node, p)
    return t
```

辅助函数 branch 和 lcp 等定义如下:

```
def maskbit(x, mask):
    return x & (~(mask - 1))

def zero(x, mask):
    return x & (mask >> 1) == 0

def lcp(p1, p2):
    diff = p1 ^ p2
    mask = 1
    while diff ≠ 0:
        diff >≥ 1
        mask <≤ 1
    return (maskbit(p1, mask), mask)

def branch(t1, t2):
    t = IntTree()
    (t.prefix, t.mask) = lcp(t1.prefix, t2.prefix)
    if zero(t1.prefix, t.mask):
        t.left, t.right = t1, t2
    else:
        t.left, t.right = t2, t1
    return t
```

图6.8展示了使用插入算法构造的前缀树。

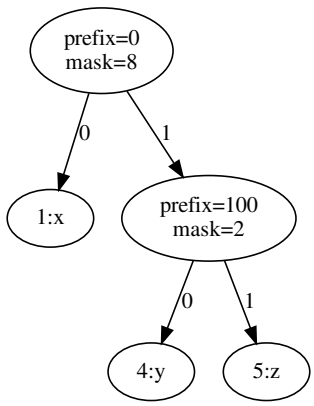


图 6.8: 插入映射 $1 \to x, 4 \to y, 5 \to z$ 到一个大端整数前缀树树

6.3.3 查找

如果前缀树为空或仅包含一个叶子节点，且节点的 key 不等于待查找的整数，则查找结果为空。如果树仅有一个叶子节点，且节点的 key 恰好等于待查找的值，则查找结果就是该叶子节点所包含的数据。否则，树 T 是一个分支节点，我们需要比较节点中存储的最长公共前缀是否和待查找的整数一致，并根据下一位是 0 还是 1 进行递归查找。如果最长公共前缀不一致，说明待查找的整数不存在。

$$lookup(T,k) = \begin{cases} \phi & : & T = \phi \vee (T = (k', v), k' \neq k) \\ v & : & T = (k', v), k' = k \\ lookup(T_l, k) & : & T = (p, m, T_l, T_r), match(k, p, m), zero(k, m) \\ lookup(T_r, k) & : & T = (p, m, T_l, T_r), match(k, p, m), \neg zero(k, m) \\ \phi & : & otherwise \end{cases} \tag{6.7}$$

下面的 Haskell 例子程序实现了递归查找算法。

```
search t k
  = case t of
      Empty → Nothing
      Leaf k' x → if k==k' then Just x else Nothing
      Branch p m l r
            | match k p m → if zero k m then search l k
                            else search r k
            | otherwise → Nothing
```

也可以用命令式方法实现查找。根据前缀树的性质，如果待查找的整数和根节点有相同的前缀，我们接下来检查前缀后的位。如果是 0，接下来在左子树查找；如果是 1，则在右子树继续查找。

当到达一个叶子节点时，我们需要比较节点的 key 是否等于待查找的整数。算法描述如下：

```
1: function LOOK-UP(T, k)
2:     if T = NIL then
3:         return NIL                                   ▷ 没找到
4:     while T is not leaf, and MATCH(k, PREFIX(T), MASK(T)) do
5:         if ZERO?(k, MASK(T)) then
6:             T ← LEFT(T)
7:         else
8:             T ← RIGHT(T)
9:     if T is leaf, and KEY(T) = k then
10:        return DATA(T)
11:    else
12:        return NIL                                   ▷ 没找到
```

下面的 Python 例子程序实现了这一查找算法。

```
def lookup(t, key):
    while t is not None and (not t.isleaf()) and t.match(key):
        if zero(key, t.mask):
            t = t.left
        else:
            t = t.right
    if t is not None and t.isleaf() and t.key == key:
        return t.value
    return None
```

6.4 字符 Trie

整数 Trie 和前缀树可以作为了解文字处理问题的起点，与其相关的技术在编译器实现中有着重要的应用。Haskell 语言的编译器 GHC（Glasgow Haskell Compiler）在 1998 年以前的实现中已经使用了整数前缀树 [21]。如果将 key 的类型扩展为字符，Trie 和前缀树就可以成为文字处理的有力武器。

6.4.1 定义

使用字符作为 key，仅仅左右两个分支就不够了。拿英语来说，一共有 26 个字符。如果忽略大小写，一种简单的办法是限定分支（子树）的个数不得超过 26。有些简化的实现使用长度为 26 的数组来管理分支。如图6.9所示。

并非所有的 26 个分支都含有数据。例如图6.9中，根节点的分支中，只有代表'a'、'b' 和'z' 的 3 个子分支不为空。其他分支，例如代表'c' 的分支，全部是空的。简单起见，我们在接下来的部分不画出这些空的分支。

如果区分大小写，或者处理英语以外的其他语言，分支的数目会超过 26。我们可以通过使用 Hash 表或者 map 等数据结构来解决动态数目分支的情况。

综上，一棵字符 Trie 或者为空，或者是一个节点。节点的类型有两种:

- 叶子节点，不含有任何子分支;
- 分支节点，含有多个子分支，每个子分支都代表一个不同的字符。

叶子节点和分支节点都可能存储相关的数据。下面的 Haskell 例子代码定义了字符 Trie。

```
data Trie a = Trie { value :: Maybe a
                   , children :: [(Char, Trie a)]}

empty = Trie Nothing []
```

下面的 ANSI C 例子代码给出了字符 Trie 的结构定义。简单起见，我们限定字符集仅仅包含小写英文字母'a' 到'z'。

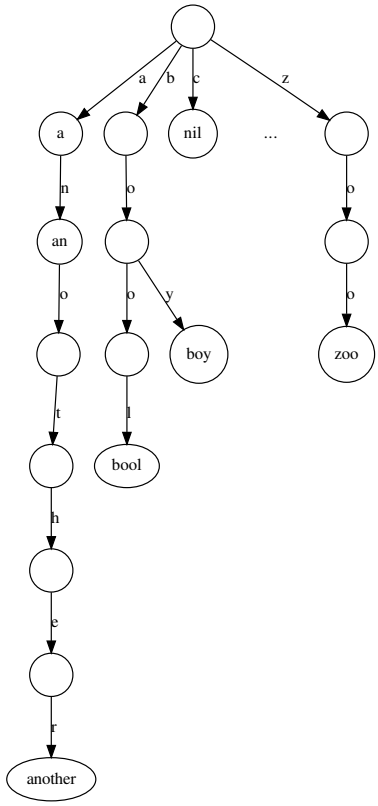


图 6.9: 最多含有 26 个分支的字符 Trie，包含 a、an、another、bool、boy 和 zoo 共 6 个 key

```
struct Trie {
    struct Trie* children[26];
    void* data;
};
```

6.4.2 插入

记待插入的字符串为 $K = k_1k_2...k_n$，其中 k_i 是第 i 个字符。K' 是除第一个字符 k_1 外的剩余字符串。v' 是待插入的数据。记树为 $T = (v, C)$，其中 v 为根节点保存的数据。$C = \{(c_1, T_1), (c_2, T_2), ..., (c_m, T_m)\}$ 为子分支的映射。它将字符 c_i 映射到子树 T_i。如果树 T 为空，则相应的映射 C 也为空。

$$insert(T, K, v') = \begin{cases} (v', C) & : & K = \phi \\ (v, ins(C, k_1, K', v')) & : & otherwise. \end{cases} \tag{6.8}$$

如果待插入的 key 为空串，我们用新数据 v' 覆盖以前的数据 v。否则，需要找到对应子分支的映射，并递归进行插入。这一过程由函数 $ins(C, k_1, K', v')$ 实现。它逐一检查 C 中的字符－子树映射对。令 C' 为除第一个映射以外的其他映射，这一函数可以定义如下：

$$ins(C, k_1, K', v') = \begin{cases} \{(k_1, insert((\phi, \phi), K', v'))\} & : & C = \phi \\ \{k_1, insert(T_1, K', v')\} \cup C' & : & k_1 = c_1 \\ \{(c_1, T_1)\} \cup ins(C', k_1, K', v') & : & otherwise \end{cases} \tag{6.9}$$

若 C 为空，我们将字符 k_1 映射到一个新的空节点上（包含一个空值和一个空子树列表），然后递归插入剩余的字符；否则算法找到字符 k_1 映射到的子树，然后递归进行插入。

下面的 Haskell 例子程序实现了这一插入算法。

```
insert t []     x = Trie (Just x)  (children t)
insert t (k:ks) x = Trie (value t) (ins (children t) k ks x) where
    ins [] k ks x = [(k, (insert empty ks x))]
    ins (p:ps) k ks x = if fst p == k
                        then (k, insert (snd p) ks x):ps
                        else p:(ins ps k ks x)
```

也可以用命令式方法实现插入。我们从根节点开始，逐一检查字符串中的每个字符和相应的分支。如果为空，就创建一个新的节点，然后处理下一个字符和对应的分支。我们重复这一过程直到处理完所有的字符。最后将数据存入此刻到达的节点。

插入算法的描述如下：

```
1: function INSERT(T, k, v)
2:     if T = NIL then
3:         T ← EMPTY-NODE
```

```
4:    p ← T
5:    for each c in k do
6:        if CHILDREN(p)[c] = NIL then
7:            CHILDREN(p)[c] ← EMPTY-NODE
8:        p ← CHILDREN(p)[c]
9:    DATA(p) ← v
10:   return T
```

下面的 ANSI C 例子程序实现了这一插入算法。

```
struct Trie* insert(struct Trie* t, const char* key, void* value) {
    int c;
    struct Trie *p;
    if(!t)
        t = create_node();
    for (p = t; *key; ++key, p = p→children[c]) {
        c = *key - 'a';
        if (!p→children[c])
            p→children[c] = create_node();
    }
    p→data = value;
    return t;
}
```

其中函数 create_node 创建一个空节点，并将所有的子分支设置为空。

```
struct Trie* create_node() {
    struct Trie* t = (struct Trie*) malloc(sizeof(struct Trie));
    int i;
    for (i = 0; i < 26; ++i)
        t→children[i] = NULL;
    t→data = NULL;
    return t;
}
```

6.4.3 查找

查找时, 我们从第一个字符开始, 如果它对应到某个子分支, 则在这个子分支上递归查找剩余的字符。记 Trie 为 (v, C), 若待查找的 key 不为空, 则记为 $K = k_1 k_2 ... k_n$。第一个字符为 k_1，剩余的字符为 K'。

$$lookup(T, K) = \begin{cases} v & : & K = \phi \\ \phi & : & find(C, k_1) = \phi \\ lookup(T', K') & : & find(C, k_1) = T' \end{cases} \tag{6.10}$$

其中函数 $find(C, k)$ 逐一检查所有的字符—子树映射对 C 以找出字符 k 对应的子树。如果映射列表 C 为空,则结果为空,查找失败。否则记 $C = \{(k_1, T_1), (k_2, T_2), ..., (k_m, T_m)\}$,

第一棵子树 T_1 对应 k_1；剩余映射对记为 C'。下面的公式定义了 $find$ 函数。

$$find(C, k) = \begin{cases} \phi & : & C = \phi \\ T_1 & : & k_1 = k \\ find(C', k) & : & otherwise \end{cases} \tag{6.11}$$

下面的 Haskell 例子程序实现了 Trie 的查找算法。它使用了标准库中提供的 `lookup` 函数。

```
find t [] = value t
find t (k:ks) = case lookup k (children t) of
                  Nothing → Nothing
                  Just t' → find t' ks
```

也可以用命令式方法实现查找。我们逐一检查待查找整数的每个字符。在子分支中找到对应的分支。当检查完最后一个字符后，当前节点中存储的数据就是查找结果。

```
1: function LOOK-UP(T, key)
2:     if T = NIL then
3:         return not found
4:     for each c in key do
5:         if CHILDREN(T)[c] = NIL then
6:             return not found
7:         T ← CHILDREN(T)[c]
8:     return DATA(T)
```

下面的 ANSI C 例子程序实现了查找算法。当查找失败时，它返回空指针 NULL。

```
void* lookup(struct Trie* t, const char* key) {
    while (*key && t && t→children[*key - 'a'])
        t = t→children[*key++ - 'a'];
    return (*key || !t) ? NULL : t→data;
}
```

练习 6.1

- 在命令式实现中，请用其他容器类数据结构来管理字符 Trie 中的子树。不同的容器类型会怎样影响性能？

6.5 字符前缀树

和整数 Trie 一样，字符 Trie 的空间利用率很低。我们可以用同样的方法将 Trie 压缩成前缀树。

6.5.1 定义

字符前缀树是一种特殊的前缀树，每个节点包含若干分支。所有的子节点拥有一个最长公共前缀串。树中不存在只含有一个子分支的节点，否则最长公共前缀的长度就可以增加，因而和“最长”的性质相矛盾。

如果把图6.9中仅含有一个子分支的节点压缩，可以得到一棵如图6.10的前缀树。

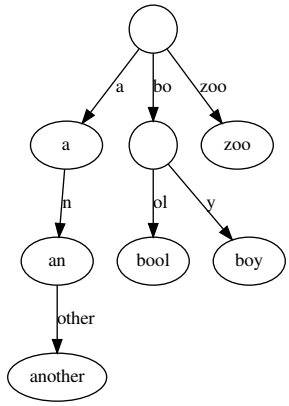


图 6.10: 一棵前缀树，含有 key：a、an、another、bool、boy 和 zoo

我们可以将字符 Trie 的定义略做修改得到前缀树的定义。一棵前缀树要么为空，要么是一个形如 $T = (v, C)$ 的节点。其中 v 代表节点中保存的附加数据；$C = \{(s_1, T_1), (s_2, T_2), ..., (s_n, T_n)\}$ 是一组映射对，每对映射包含一个字符串 s_i 和对应的子树 T_i。

下面的 Haskell 例子代码定义了前缀树树。

```
data PrefixTree k v = PrefixTree { value :: Maybe v
                                 , children :: [([k], PrefixTree k v)]}

empty = PrefixTree Nothing []

leaf x = PrefixTree (Just x) []
```

下面的 Python 例子代码重用了 Trie 来定义前缀树。

```
class PrefixTree:
    def __init__(self, value = None):
        self.value = value
        self.subtrees = {}
```

6.5.2 插入

插入字符串 s 时，若树为空，则创建一个叶子节点，如图6.11（a）所示。否则，我们逐一检查子分支映射。如果存在某个子分支 T_i 对应到字符串 s_i，并且 s_i 和 s 存在共同的前缀，我们分叉出一个新的分支 T_j。具体来说，我们创建一个新的内部分支

节点，将其映射到公共前缀，然后将 T_i 和 T_j 作为新节点的两棵子树。T_i 和 T_j 共有这一前缀。如图6.11（b）所示。这里存在两种特殊情况：一种是 s 为 s_i 的前缀，如图6.11（c）所示；另外一种是 s_i 为 s 的前缀，如图6.11（d）所示。

记前缀树为 $T = (v, C)$，函数 $insert(T, k, v')$ 将字符串 k 和数据 v' 插入到 T 中。

$$insert(T, k, v') = (v, ins(C, k, v')) \tag{6.12}$$

这里，我们调用另外一个函数 $ins(C, k, v')$ 还实现插入。如果子分支的映射 C 为空,我们创建一个新叶子节点;否则需要逐一检查每个子树。记 $C = \{(k_1, T_1), (k_2, T_2), ..., (k_n, T_n)\}$，$C'$ 为除去第一个“前缀—映射”对以外的所有其他映射。

$$ins(C, k, v') = \begin{cases} \{(k, (v', \phi))\} & : & C = \phi \\ \{(k, (v', C_{T_1}))\} \cup C' & : & k_1 = k \\ \{branch(k, v', k_1, T_1)\} \cup C' & : & match(k_1, k) \\ \{(k_1, T_1)\} \cup ins(C', k, v') & : & otherwise \end{cases} \tag{6.13}$$

第一行处理映射为空的边界情况。我们创建一个叶子节点，将 v' 存入其中，然后将 k 映射到这个节点上。并返回这一映射对。第二行处理待插入的 key 已经存在的情况，我们用新数据 v' 覆盖了原来的数据。其中 C_{T_1} 表示子树 T_1 的所有子分支映射。第三行处理 k 和第一个映射对中的 key 匹配的情况。最后一行继续查找剩余的子树映射。

我们称两个串 A 和 B 匹配当且仅当它们含有非空的公共前缀。

$$match(A, B) = A \neq \phi \wedge B \neq \phi \wedge a_1 = b_1 \tag{6.14}$$

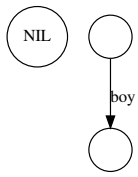
其中 a_1 和 b_1 分别是 A 和 B 不为空时的第一个字符。

函数 $branch(k_1, v, k_2, T_2)$ 的参数包括两个串 k_1 和 k_2，数据 v 和一棵树 T_2。它查找两个串的最长公共前缀 $k = lcp(k_1, k_2)$，记前缀后不同的部分为：$k_1' = k_1 - k$ 和 $k_2' = k_2 - k$。我们首先要处理两种边界情况：k_1 为 k_2 的前缀，或者 k_2 为 k_1 的前缀。对于第一种情况，我们创建一个新的叶子节点，将 v 存入其中，将 k 映射到这个节点上。然后令 (k_2', T_2) 为唯一的子树映射。对于第二种情况，我们递归地将 k_1 和 v 插入 T_2。否则，我们需要创建一个分支节点，将其作为最长公共前缀 k 的映射。这一分支节点有两个子树，一个是 (k_2', T_2), 另外一个是一个叶子节点，存有数据 v，并且是 k_1' 的映射。

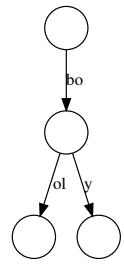
$$branch(k_1, v, k_2, T_2) = \begin{cases} (k, (v, \{(k_2', T_2)\})) & : & k = k_1 \\ (k, insert(T_2, k_1', v)) & : & k = k_2 \\ (k, (\phi, \{(k_1', (v, \phi)), (k_2', T_2)\}) & : & otherwise \end{cases} \tag{6.15}$$

其中

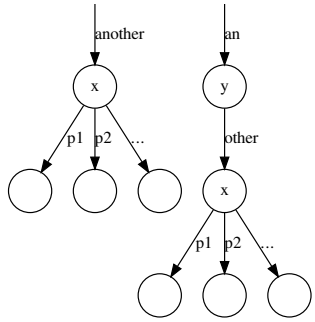
$$\begin{aligned} k &= lcp(k_1, k_2) \\ k_1' &= k_1 - k \\ k_2' &= k_1 - k \end{aligned}$$



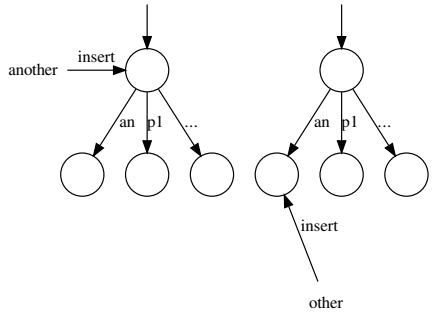
(a) 将字符串 boy 插入一空树，结果为一叶子节点。



(b) 继续插入 bool，创建一新分支，对应的公共前缀为 bo。



(c) 以字符串 an 作为 key 将数据 y 插入。根节点存有数据 x，并对应前缀 another。



(d) 某一子分支对应前缀 an，将字符串 another 作为 key 插入。需要递归将子串 other 插入到子分支中。

图 6.11: 插入前缀树树的各种情况

函数 $lcp(A, B)$ 不断从 A 和 B 中提取相同的字符。记 a_1 和 b_1 分别为 A 和 B 非空时的第一个字符，A' 和 B' 代表剩余的字符。

$$lcp(A, B) = \begin{cases} \phi & : \quad A = \phi \vee B = \phi \vee a_1 \neq b_1 \\ \{a_1\} \cup lcp(A', B') & : \quad a_1 = b_1 \end{cases} \tag{6.16}$$

下面的 Haskell 例子程序实现了前缀树的插入算法。

```
import Data.List (isPrefixOf)

insert :: Eq k ⇒ PrefixTree k v → [k] → v → PrefixTree k v
insert t ks x = PrefixTree (value t) (ins (children t) ks x) where
    ins []     ks x = [(ks, leaf x)]
    ins (p@(ks', t') : ps) ks x
        | ks' == ks
            = (ks, PrefixTree (Just x) (children t')) : ps  -- overwrite
        | match ks' ks
            = (branch ks x ks' t') : ps
        | otherwise
            = p : (ins ps ks x)

match x y = x ≠ [] && y ≠ [] && head x == head y

branch :: Eq k ⇒ [k] → v → [k] → PrefixTree k v → ([k], PrefixTree k v)
branch ks1 x ks2 t2
    | ks1 == ks
        -- ex: insert "an" into "another"
        = (ks, PrefixTree (Just x) [(ks2', t2)])
    | ks2 == ks
        -- ex: insert "another" into "an"
        = (ks, insert t2 ks1' x)
    | otherwise = (ks, PrefixTree Nothing [(ks1', leaf x), (ks2', t2)])
   where
      ks = lcp ks1 ks2
      m = length ks
      ks1' = drop m ks1
      ks2' = drop m ks2

lcp :: Eq k ⇒ [k] → [k] → [k]
lcp [] _ = []
lcp _ [] = []
lcp (x:xs) (y:ys) = if x==y then x : (lcp xs ys) else []
```

命令式的插入算法描述如下：

```
1: function INSERT(T, k, v)
2:     if T = NIL then
3:         T ← EMPTY-NODE
4:     p ← T
5:     loop
6:         match ← FALSE
```

```
    for each (s_i, T_i) ∈ CHILDREN(p) do
        if k = s_i then
            VALUE(T_i) ← v                                  ▷ 覆盖
            return T
        c ← LCP(k, s_i)
        k_1 ← k − c
        k_2 ← s_i − c
        if c ≠ NIL then
            match ← TRUE
            if k_2 = NIL then                               ▷ s_i 是 k 的前缀
                p ← T_i
                k ← k_1
                break
            else                                            ▷ 分支出一个新叶子节点
                ADD(CHILDREN(p), (c, BRANCH(k_1, LEAF(v), k_2, T_i)))
                DELETE(CHILDREN(p), (s_i, T_i))
                return T
    if ¬match then                                          ▷ 增加一个新叶子节点
        ADD(CHILDREN(p), (k, LEAF(v)))
        break
27: return T
```

上述算法中，函数 LCP 寻找两个字符串的最长公共前缀。例如字符串 bool 和 boy 的最长公共前缀为 bo。字符串的减号（-）运算用以给出两个字符串的不同部分。例如 bool - bo = ol。函数 BRANCH 负责创建分支节点并更新对应的 key。

为了获取最长公共前缀，我们可以逐一比较两个字符串的字符，直到遇到不相同的字符为止。

```
1: function LCP(A, B)
2:     i ← 1
3:     while i ≤ |A| ∧ i ≤ |B| ∧ A[i] = B[i] do
4:         i ← i + 1
5:     return A[1...i − 1]
```

分支出新叶子节点时存在两种情况。$\text{BRANCH}(s_1, T_1, s_2, T_2)$ 的参数是两个不同的 key 和两棵树。如果 s_1 为空，说明一个字符串是另一个的前缀。例如把字符串 an 插入到一棵前缀为 another 的树中。这种情况结果是 T_2 成为了 T_1 的一棵子树。否则，如果 s_1 不为空，我们需要创建出一个新的分支节点，并令 T_1 和 T_2 分别为新节点的两棵子树。

```
1: function BRANCH(s_1, T_1, s_2, T_2)
```

2: **if** $s_1 = \phi$ **then**
3: ADD(CHILDREN(T_1), (s_2, T_2))
4: **return** T_1
5: $T \leftarrow$ EMPTY-NODE
6: CHILDREN(T) $\leftarrow \{(s_1, T_1), (s_2, T_2)\}$
7: **return** T

下面的 Python 例子程序实现了前缀树的插入算法。

```
def insert(t, key, value = None):
    if t is None:
        t = PrefixTree()
    node = t
    while True:
        match = False
        for k, tr in node.children.items():
            if key == k: #覆盖原先数据
                tr.value = value
                return t
            prefix, k1, k2 = lcp(key, k)
            if prefix ≠ "":
                match = True
                if k2 == "":
                    #例如将 “another” 插入前缀为 “an” 的树，继续遍历
                    node = tr
                    key = k1
                    break
                else: #分支出一个新叶子节点
                    node.children[prefix] = branch(k1, Patricia(value), k2, tr)
                    del node.children[k]
                    return t
        if not match: #增加一个新叶子节点
            node.children[key] = PrefixTree(value)
            return t
    return t
```

其中查找最长公共前缀和分支出新节点的函数实现如下：

```
#返回 (p, s1', s2'), 其中 p 是 lcp, s1'=s1-p, s2'=s2-p
def lcp(s1, s2):
    j = 0
    while j < len(s1) and j < len(s2) and s1[j] == s2[j]:
        j += 1
    return (s1[0:j], s1[j:], s2[j:])

def branch(key1, tree1, key2, tree2):
    if key1 == "":
        #例如将 “an” 插入到前缀为 “another” 的树中
        tree1.children[key2] = tree2
        return tree1
    t = PrefixTree()
    t.children[key1] = tree1
```

```
    t.children[key2] = tree2
    return t
```

6.5.3 查找

和 Trie 不同，我们不能逐一根据每个字符查找。我们从根节点开始，检查子分支中是否存在某个子树对应的 key 是待查找字符串的前缀。如果存在，我们从待查找串中将这个前缀去掉，然后在这棵子树中递归查找；否则，如果没有任何子树对应到待查找串的前缀，则查找失败。

记前缀树为 $T = (v, C)$，下面的定义调用 $find$ 函数在所有子分支 C 中进行查找。

$$lookup(T, k) = find(C, k) \tag{6.17}$$

若 C 为空，则查找失败，否则记 $C = \{(k_1, T_1), (k_2, T_2), ..., (k_n, T_n)\}$，我们首先检查 k 是否是 k_1 的前缀，如果不是，就递归地在剩余的映射对 C' 中查找。

$$find(C, k) = \begin{cases} \phi & : & C = \phi \\ v_{T_1} & : & k = k_1 \\ lookup(T_1, k - k_1) & : & k_1 \sqsubset k \\ find(C', k) & : & otherwise \end{cases} \tag{6.18}$$

其中 $A \sqsubset B$ 表示 A 为 B 的前缀，如果某个子分支对应的 key 是 k 的前缀，则 $find$ 函数就相互递归（mutually recursive call）地调用 $lookup$ 函数进行查找。

下面的 Haskell 例子程序实现了查找算法。

```
find :: Eq k ⇒ PrefixTree k v → [k] → Maybe v
find t = find' (children t) where
    find' [] _ = Nothing
    find' (p@(ks', t') : ps) ks
          | ks' == ks = value t'
          | ks' `isPrefixOf` ks = find t' (diff ks ks')
          | otherwise = find' ps ks
    diff ks1 ks2 = drop (length (lcp ks1 ks2)) ks1
```

查找算法也可以用命令式的方法实现。

```
1: function LOOK-UP(T, k)
2:     if T = NIL then
3:         return not found
4:     repeat
5:         match ← FALSE
6:         for ∀(k_i, T_i) ∈ CHILDREN(T) do
7:             if k = k_i then
8:                 return DATA(T_i)
```

```
9:            if k_i is prefix of k then
10:               match ← TRUE
11:               k ← k − k_i
12:               T ← T_i
13:               break
14:       until ¬match
15:       return not found
```

下面的 Python 例子程序实现了查找算法。它复用了前面定义的 `lcp(s1, s2)` 函数来检查一个字符串是否是另一个的前缀。

```
def lookup(t, key):
    if t is None:
        return None
    while True:
        match = False
        for k, tr in t.subtrees.items():
            if k == key:
                return tr.value
            prefix, k1, k2 = lcp(key, k)
            if prefix ≠ "" and k2 == "":
                match = True
                key = k1
                t = tr
                break
        if not match:
            break
    return None
```

6.6 Trie 和前缀树的应用

Trie 和前缀树可以用来解决许多有趣的问题。本节中，我们给出一些例子，包括电子词典，单词自动补齐，T9 输入法等。

6.6.1 电子词典和单词自动补齐

图6.12展示的是某英汉电子词典的界面。为了易用，当用户输入某些字符后，电子词典会搜索词库，将所有候选单词全部列出。

电子词典通常存有数十万单词。进行全词查找的开销很大。商业电子词典软件会同时使用多重方法以提高性能，包括缓存（caching）、索引（indexing）等等。

和电子词典类似，图6.13显示了某互联网搜索引擎的界面。当用户输入内容后，会列出一些可能的候选搜索项。这些项的开头部分和用户输入相匹配[1]，并且按照被

[1]实际功能会更加复杂，包括拼写检查，关键词提取、引导等。

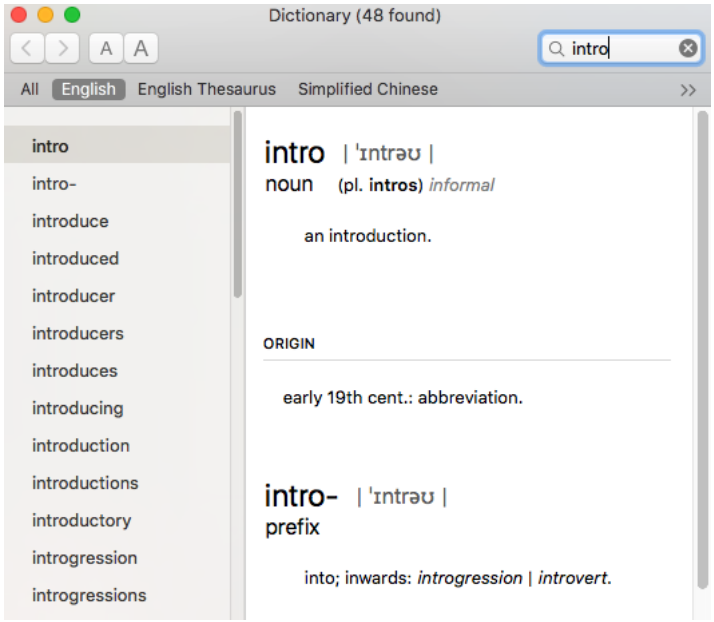


图 6.12: 电子词典。所有和用户输入匹配的候选单词全被列出

搜索的热门程度排序。被搜索的次数越多，越排在前面。

这两个例子中，软件都提供了某种自动完成的机制。在某些现代的 IDE（集成开发环境）中，编辑器还可以帮助用户自动完成程序代码。

我们看看如何使用前缀树来实现电子词典。为了简化问题，假设我们的词典是英—英词典。

词典中保存了 key-value 对，key 是英文单词或者词组，value 是对应的解释。

当用户输入‘a’的时候，词典不是只给出‘a’的意思，而是提供一系列候选单词的列表。这些候选单词都以‘a’开头，包括 abandon、about、accent、adam……当然，这些都是存储在前缀树中的单词。

如果候选单词太多，一种方案是只显示前 10 个，如果用户查找的单词不在其中，他可以浏览更多的候选项。如果待查找的字符串为空，我们从当前节点扩展出前 n 个子节点作为候选项；否则，我们递归地在有共同前缀的子分支中查找。

在支持惰性求值（lazy evaluation）的编程环境中，一种简单直观的方法是惰性扩展全部的子节点，然后根据需要取前 n 个。记前缀树前缀树为 $T = (v, C)$，下面的函数枚举所有以 k 开头的内容。

$$findAll(T, k) = \begin{cases} enum(C) & : & k = \phi, v = \phi \\ \{(\phi, v)\} \cup enum(C) & : & k = \phi, v \neq \phi \\ find(C, k) & : & k \neq \phi \end{cases} \tag{6.19}$$

前两行处理 key 为空的边界情况。此时，我们通过枚举扩展所有数据不为空的子节点。最后一行调用 $find$ 函数寻找和前缀 k 匹配的子分支。

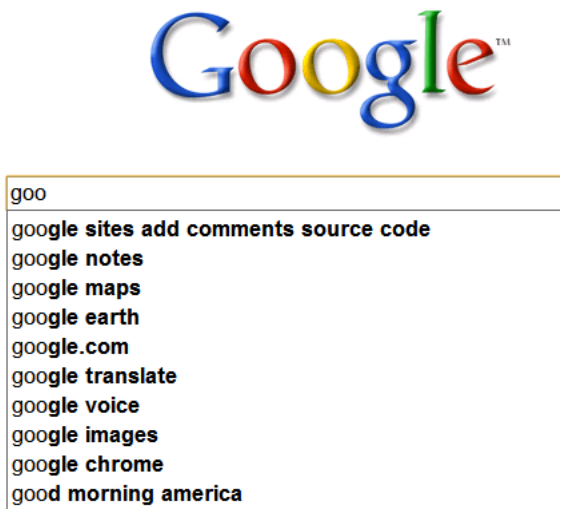


图 6.13: 搜索引擎。和用户输入匹配的候选搜索被列出

如果节点的子分支不为空，记 $C = \{(k_1, T_1), (k_2, T_2), ..., (k_m, T_m)\}$，令除去第一对映射以外的剩余映射为 C'。枚举算法可以定义如下：

$$enum(C) = \begin{cases} \phi & : \quad C = \phi \\ mapAppend(k_1, findAll(T_1, \phi)) \cup enum(C') & : \end{cases} \tag{6.20}$$

其中 $mapAppend(k, L) = \{(k + k_i, v_i) | (k_i, v_i) \in L\}$。它将前缀 k 添加到列表 L 中的所有 key-value 对的 key 前面[2]。

函数 $enum$ 也可用 $concatMap$ 的概念定义（亦称为 $flatMap$）[3]。

$$enum(C) = concatMap(\lambda_{(k,T)}.mapAppend(k, findAll(T, \phi))) \tag{6.21}$$

函数 $find(C, k)$ 定义如下。如果子树为空，结果也为空；否则，它首先检查 k_1 映射的子树 T_1。如果 k_1 和 k 相等，就调用 $mapAppend$ 向 T_1 所有子分支的 key 前增加前缀 k；如果 k_1 是 k 的前缀，算法就递归地查找所有以 $k - k_1$ 开头的子分支；反之，如果 k 是 k_1 的前缀，则 T_1 的所有子分支都是候选项。否则，算法跳过第一对映射，继续处理剩余的其他映射。

$$find(C, k) = \begin{cases} \phi & : \quad C = \phi \\ mapAppend(k_1, findAll(T_1, \phi)) & : \quad k \sqsubset k_1 \\ mapAppend(k_1, findAll(T_1, k - k_1)) & : \quad k_1 \sqsubset k \\ find(C', k) & : \quad otherwise \end{cases} \tag{6.22}$$

下面的 Haskell 例子程序按照上述算法实现了一个简单的电子词典：

[2]可以进一步抽象为对第一个元素进行映射。在 Haskell 中，使用范畴论中的箭头概念，$mapAppend$ 可被表示为 $map(first(k+), L)$

[3]效果上相当于先对每个元素进行映射，然后将结果连接起来。通常使用 build-foldr 来消除中间结果中的列表

```
import Control.Arrow (first)

get n t k = take n $ findAll t k

findAll :: Eq k ⇒ PrefixTree k v → [k] → [([k], v)]
findAll (PrefixTree Nothing cs) [] = enum cs
findAll (PrefixTree (Just x) cs) [] = ([], x) : enum cs
findAll (PrefixTree _ cs) k = find' cs k
  where
    find' [] _ = []
    find' ((k', t') : ps) k
          | k `isPrefixOf` k'
              = map (first (k' ++)) (findAll t' [])
          | k' `isPrefixOf` k
              = map (first (k' ++)) (findAll t' $ drop (length k') k)
          | otherwise = find' ps k

enum :: Eq k ⇒ [([k], PrefixTree k v)] → [([k], v)]
enum = concatMap (λ(k, t) → map (first (k ++)) (findAll t []))
```

在 Haskell 这样的惰性求值编程环境中,前 n 个候选项可以通过 $take(n, findAll(T, k))$ 来获取。附录 A 给出了 $take$ 函数的详细定义。

也可以用命令式的方式实现电子词典。下面的算法复用了前面定义的前缀树查找函数。当我们找到一个节点，其对应的前缀和用户输入的内容一致时，算法将扩展该节点的所有子树直到获取到前 n 个候选项。

```
1: function LOOK-UP(T, k, n)
2:     if T = NIL then
3:         return φ
4:     prefix ← NIL
5:     repeat
6:         match ← FALSE
7:         for ∀(k_i, T_i) ∈ CHILDREN(T) do
8:             if k is prefix of k_i then
9:                 return EXPAND(prefix + k_i, T_i, n)
10:            if k_i is prefix of k then
11:                match ← TRUE
12:                k ← k − k_i
13:                T ← T_i
14:                prefix ← prefix + k_i
15:                break
16:    until ¬match
17:    return φ
```

其中函数 EXPAND$(T, prefix, n)$ 选取 n 个子树，这些子树在 T 中有同样的前缀。它的实现为广度优先遍历（BFS），本书最后一章对包括广度优先在内的搜索算法有详细的介绍。

1: **function** EXPAND$(prefix, T, n)$
2: $R \leftarrow \phi$
3: $Q \leftarrow \{(prefix, T)\}$
4: **while** $|R| < n \wedge Q$ is not empty **do**
5: $(k, T) \leftarrow$ POP(Q)
6: **if** DATA$(T) \neq$ NIL **then**
7: $R \leftarrow R \cup \{(k,$ DATA$(T)\ )\}$
8: **for** $\forall (k_i, T_i) \in$ CHILDREN(T) in sorted order **do**
9: PUSH$(Q, (k + k_i, T_i))$

下面的 Python 例子程序实现了一个电子词典。它使用了标准库中的 `find` 函数来判断一个字符串是否是另一个的前缀。

```
def lookup(t, key, n):
    if t is None:
        return []
    prefix = ""
    while True:
        match = False
        for k, tr in t.subtrees.items():
            if string.find(k, key) == 0: # key is prefix of k
                return expand(prefix + k, tr, n)
            if string.find(key, k) ==0:
                match = True
                key = key[len(k):]
                t = tr
                prefix += k
                break
        if not match:
            break
    return []

def expand(prefix, t, n):
    res = []
    q = [(prefix, t)]
    while len(res)<n and q:
        (s, p) = q.pop(0)
        if p.value is not None:
            res.append((s, p.value))
        for k, tr in sorted(p.subtrees.items()):
            q.append((s + k, tr))
    return res
```

6.6.2 T9 输入法

手机用户编辑短信或者电子邮件时的体验和 PC 上完全不同。手机键盘（称为 ITU-T 键盘）上只有非常少的按键。如图6.14所示。

图 6.14: 手机 ITU-T 键盘

在 ITU-T 键盘上输入英文单词或短语有两种方法。例如用户要输入单词 home，他需要按照下面的顺序按键：

- 按两次 4 键以输入字符 h；
- 按三次 6 键以输入字符 o；
- 按一次 6 键以输入字符 m；
- 按两次 3 键以输入字符 e；

另外一种更快速的方法使用下面的按键顺序：

- 依次按下 4、6、6、3，单词 home 出现在候选列表的最上方；
- 按下‘*’号键以变换不同的候选单词，good 此时出现在候选列表上；
- 按下‘*’号键再次变换，下一个候选单词 gone 出现在列表上；
- ……

对比这两个方法，可以发现后者更加方便，但是需要额外保存一个候选单词字典。这种方法被称作“T9 输入法”或预测输入法 [25]、[26]。T9 是英文 textonym 的缩写，它以 T 开头，后面跟 9 个字母。T9 输入法可以用前缀树来实现。

为了向用户提供候选单词，T9 输入法需要预先准备一个词典。商业上的 T9 输入法通常使用更加复杂的索引词典，同时在文件系统和缓存中进行快速索引。本节中的实现仅仅出于演示的目的。

首先我们需要定义 T9 键盘映射，它将数字映射为候选字符。

$$\begin{aligned} M_{T9} = \{ \quad & 2 \to abc, 3 \to def, 4 \to ghi, \\ & 5 \to jkl, 6 \to mno, 7 \to pqrs, \\ & 8 \to tuv, 9 \to wxyz\} \end{aligned} \tag{6.23}$$

使用这一映射后，$M_{T9}[i]$ 就返回数字 i 对应的若干个字符。我们也可以定义从字符到数字的逆映射。

$$M_{T9}^{-1} = concat(\{\{c \to d | c \in S\} | (d \to S) \in M_{T9}\}) \tag{6.24}$$

通过查找 M_{T9}^{-1}，我们可以将字符串转换成一组按键序列。

$$digits(S) = \{M_{T9}^{-1}[c] | c \in S\} \tag{6.25}$$

给定一个数字序列 $D = d_1 d_2 ... d_n$，我们定义 T9 查找的算法如下。

$$findT9(T, D) = \begin{cases} \{\phi\} & : \quad D = \phi \\ concatMap(find, prefixes(T)) & : \quad otherwise \end{cases} \tag{6.26}$$

其中 T 是从一组单词和词组中构造出的前缀树。我们将其作为 T9 输入法的词典。若输入的数字串 D 为空，则候选单词的结果列表也是空。否则，我们搜索匹配输入的子树并将结果连接起来。

为了枚举所有匹配的子树，我们检查子树集 C_T 中的每一对 (k_i, T_i)。首先将字符串 k_i 转换为数字串 d_i，然后比较 d_i 和 D。如果其中之一是另一个的前缀，则选中这一对 (k_i, T_i) 进行进一步查找。

$$prefixes(T) = \{(k_i, T_i) | (k_i, T_i) \in C_T, d_i = digits(k_i), d_i \sqsubset D \vee D \sqsubset d_i\} \tag{6.27}$$

函数 $find$ 接受两个参数，一个前缀 S 和需要进一步查找的子树 T'。其中 S 是 D 的前缀。为此，我们从 D 中去除这一前缀，然后用剩余的数字串 $D' = D - S$ 继续查找。最后再将 S 添加回递归查找的各个结果前面。

$$find(S, T') = \{take(n, S + s_i) | s_i \in findT9(T', D - S)\} \tag{6.28}$$

其中 $n = |D|$ 是输入数字串的长度。函数 $take(n, L)$ 从列表 L 中获取前 n 个元素。如果列表的长度短于 n，则获取全部元素。

下面的 Haskell 例子程序实现了从前缀树查找的 T9 输入法。

```
import qualified Data.Map as Map

mapT9 = Map.fromList [('1', ",."), ('2', "abc"), ('3', "def"), ('4', "ghi"),
                      ('5', "jkl"), ('6', "mno"), ('7', "pqrs"), ('8', "tuv"),
                      ('9', "wxyz")]

rmapT9 = Map.fromList $ concatMap (λ(d, s) → [(c, d) | c ← s]) $ Map.toList mapT9
```

```
digits = map (λc → Map.findWithDefault '#' c rmapT9)

findT9 :: PrefixTree Char v → String → [String]
findT9 t [] = [""]
findT9 t k = concatMap find prefixes
  where
    n = length k
    find (s, t') = map (take n ∘ (s++)) $ findT9 t' (k `diff` s)
    diff x y = drop (length y) x
    prefixes = [(s, t') | (s, t') ← children t, let ds = digits s in
                          ds `isPrefixOf` k || k `isPrefixOf` ds]
```

可以使用广度优先搜索实现命令式的 T9 输入法。我们使用一个队列 Q，其保存的元素为三元组 $(prefix, D, T)$。每个三元组包含一个已搜索过的前缀 $prefix$，尚未搜索的剩余部分 D，还有待搜索的子树 T。队列初始的时候，三元组包含一个空前缀，全部待搜索的数字，以及前缀树的根节点。算法不断从队列中取出三元组，对其中的树 T，依次检查它的子树 T_i。通过检索 T9 的逆映射，我们将子树对应的前缀串 k_i 转换成数字串 D'。如果 D 是 D' 的前缀，说明找到了一个候选单词。我们将 k_i 添加到三元组中的前缀 $prefix$ 后，并纪录下这一结果。如果 D' 是 D 的前缀，我们需要继续在这棵子树中查找。为此，我们将已 k_i 结尾的前缀，剩余的数字串 $D - D'$ 和该子树组成一个新的三元组，放入队列的尾部。

1: **function** LOOK-UP-T9(T, D)
2: $R \leftarrow \phi$
3: **if** $T =$ NIL or $D = \phi$ **then**
4: **return** R
5: $n \leftarrow |D|$
6: $Q \leftarrow \{(\phi, D, T)\}$
7: **while** $Q \neq \phi$ **do**
8: $(prefix, D, T) \leftarrow$ POP(Q)
9: **for** $\forall (k_i, T_i) \in$ CHILDREN(T) **do**
10: $D' \leftarrow$ DIGITS(k_i)
11: **if** $D' \sqsubset D$ **then** ▷ D' is prefix of D
12: $R \leftarrow R \cup \{$ TAKE $(n, prefix + k_i)\}$ ▷ 限制长度不超过 n
13: **else if** $D \sqsubset D'$ **then**
14: PUSH($Q, (prefix + k_i, D - D', T_i)$)
15: **return** R

函数 DIGITS(S) 将字符串 S 转换为数字串。

1: **function** DIGITS(S)
2: $D \leftarrow \phi$
3: **for** each $c \in S$ **do**
4: $D \leftarrow D \cup \{M_{T9}^{-1}[c]\}$

5: **return** D

下面的 Python 例子程序使用前缀树实现了 T9 输入法。

```
T9MAP={'2':"abc", '3':"def", '4':"ghi", '5':"jkl", λ
       '6':"mno", '7':"pqrs", '8':"tuv", '9':"wxyz"}

T9RMAP = dict([(c, d) for d, cs in T9MAP.items() for c in cs])

def digits(w):
    return ''.join([T9RMAP[c] for c in w])

def lookup_t9(t, key):
    if t is None or key == "":
        return []
    res = []
    n = len(key)
    q = [("", key, t)]
    while q:
        prefix, key, t = q.pop(0)
        for k, tr in t.subtrees.items():
            ds = digits(k)
            if string.find(ds, key) == 0: # key is prefix of ds
                res.append((prefix + k)[:n])
            elif string.find(key, ds) == 0: # ds is prefix of key
                q.append((prefix + k, key[len(k):], tr))
    return res
```

练习 6.2

- 使用 Trie 实现电子词典和 T9 输入法。
- 对于返回多个候选结果的前缀树查找算法，如何保证输出的结果按照字典顺序排序？这会对性能产生怎样的影响？
- 如果不使用惰性求值，如何实现函数式的电子词典和 T9 输入法？

6.7 小结

本章一开始介绍了整数 Trie 和前缀树。基于整数前缀树的映射结构在编译器的实现中得到了重要的应用。字符 Trie 和字符前缀树可以看做是整数映射结构的自然扩展。它们可以被用来处理文字信息。作为例子，我们介绍了自动预测完成输入的电子词典和 T9 输入法。尽管和商业软件的实现不同，这些例子展示了如何使用 Trie 和前缀树来解决问题的方法。某些重要的数据结构，如后缀树和本章中介绍的内容紧密相关。读者可以参考附录 D。

第七章　B 树

7.1　简介

B 树是一个重要的数据结构。它常被用于解决磁盘或者外部存储器中整块数据的存取 [4]，现代文件系统有许多是由 B 树的扩展形式 B+ 树实现的。B 树还被广泛用于数据库的实现。我们可以把平衡二叉树的概念进行抽象、推广，从而引出 B 树 [39]。

图7.1展示了一棵 B 树，我们可以观察它和二叉搜索树之间的异同。

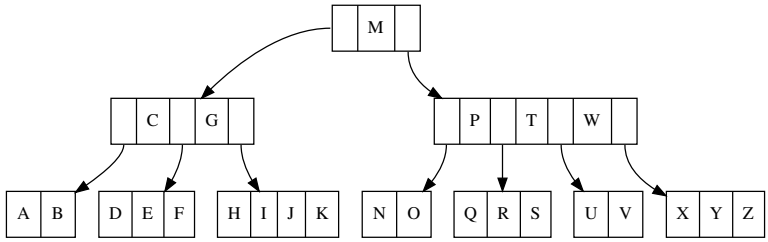


图 7.1: B 树

回忆一下二叉搜索树的定义。一棵二叉搜索树：

- 或者是一个空节点；
- 或者包含三部分，一个键值，一棵左侧分支和一棵右侧分支。这两个子分支也都是二叉搜索树。

同时，二叉搜索树满足下面的限制条件：

- 任何左侧分支中的键值都不大于节点的键值；
- 节点的键值不大于任何右侧分支中的键值。

对于非空的二叉树 (L, k, R)，其中 L、R 和 k 分别代表左右子树和键值。若函数 $Key(T)$ 可以获取树 T 的键值，这一限制条件可以表示为如下形式：

$$\forall x \in L, \forall y \in R \Rightarrow Key(x) \leq k \leq Key(y) \tag{7.1}$$

将这一定义推广，如果树中含有多个键值和子分支，它就是一棵 B 树。定义如下：

一棵 B 树

- 或者为空；
- 或者包含 n 个键值和 $n+1$ 棵子树，每棵子树也是一棵 B 树。这些键值和子树分别记为 $k_1, k_2, ..., k_n$ 和 $c_1, c_2, ..., c_n, c_{n+1}$。

图7.2描述了一个 B 树节点的样子。

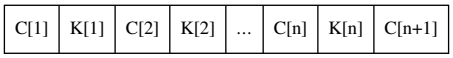


图 7.2: 一个 B 树节点

节点中的所有键值和子树都满足下面的限制条件：

- 所有的键值按照单调增（非递减）的顺序保存。即：$k_1 \leq k_2 \leq ... \leq k_n$；
- 对于任意 k_i，子树 c_i 中所有的元素都不大于 k_i，且 k_i 不大于子树 c_{i+1} 的任意元素。

这一限制条件可以表达为下面的式 (7.2)。

$$\forall x_i \in c_i, i = 0, 1, ..., n, \Rightarrow x_1 \leq k_1 \leq x_2 \leq k_2 \leq ... \leq x_n \leq k_n \leq x_{n+1} \tag{7.2}$$

最后，为了保证平衡性，B 树还满足一些额外的要求：

- 所有的叶子节点具有相同的深度；
- 定义整数 t，称为 B 树的最小度数：
  - 每个节点最多含有 $2t-1$ 个键值；
  - 除根节点外，每个节点最少含有 $t-1$ 个键值。

考虑一棵含有 n 个键值的 B 树，最小度数 $t \geq 2$，树的高度为 h。除根节点外的全部节点至少含有 $t-1$ 个键值。因为根节点至少含有一个键值，所以至少有两个深度为 1 的子节点，至少有 $2t$ 个深度为 2 的子节点，至少有 $2t^2$ 个深度为 3 的子节点……最后，至少有 $2t^{h-1}$ 个深度为 h 的叶子节点。除根节点外，将节点个数乘以 $t-1$，就可以得到 B 树中存储的全部元素个数。它必然满足下面的不等式。

$$\begin{aligned} n &\geq 1 + (t-1)(2 + 2t + 2t^2 + ... + 2t^{h-1}) \\ &= 1 + 2(t-1)\sum_{k=0}^{h-1} t^k \\ &= 1 + 2(t-1)\frac{t^h - 1}{t-1} \\ &= 2t^h - 1 \end{aligned} \tag{7.3}$$

于是可以导出 B 树的高度和元素数满足下面的关系：

$$h \leq \log_t \frac{n+1}{2} \tag{7.4}$$

这就证明了 B 树的平衡性。最简单的 B 树称为 2-3-4 树。它的最小度数 $t = 2$，除根节点外的任何节点都包含 2 到 4 个键值。任何一棵红黑树本质上都可以转换为一棵 2-3-4 树。

下面的 Python 例子代码给出了 B 树的定义。它根据传入的最小度数 t 创建一个节点：

```
class BTree:
    def __init__(self, t):
        self.t = t
        self.keys = []
        self.children = []
```

B 树的节点通常还保存有额外的数据（卫星数据），为了简化问题，我们暂时不考虑这些额外数据。

本章中，我们首先介绍如何通过插入操作构造 B 树。为了保持平衡，我们会介绍一种方法 [4]，将过满的节点在插入前进行拆分；此外，我们会介绍另外一种和红黑树类似的方法，它采用先插入后调整的策略 [13] [39]。最后，我们会介绍 B 树的删除和查找算法。

7.2 插入

我们可以通过不断插入 key 来构建 B 树。方法和二叉搜索树类似。当插入 x 时，从根节点开始，我们在节点中找到一个位置，这个位置左侧的所有 key 都小于 x，而右侧的所有 key 都大于 x[1]。如果当前节点是叶子节点，并且没有满（节点中含有的 key 不足 $2t-1$ 个），就可以将 x 插入到这个位置。否则，这一位置会指向一个子节点，我们需要递归向这一子节点插入 x。

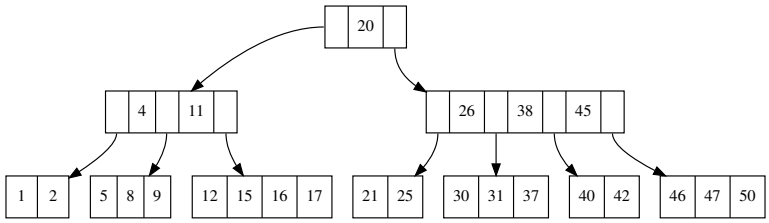
图7.3描述了一个插入的例子。这里的 B 树为 2-3-4 树。当插入元素 $x = 22$ 时，由于它的比根节点保存的 key 大，所以接下来检查右侧节点中的 26、38 和 45；因为 $22 < 26$，所以接下来检查第一个子节点中的 21 和 25。这是一个叶子节点，并且未满。因此 22 被插入到 21 和 25 中间。

但是，如果叶子节点中已经含有 $2t-1$ 个 key，它已经满了。我们就不能简单地将新 key 插入。对于图中的 B 树，插入 18 就会遇到这个问题。有两种方法可以解决它。

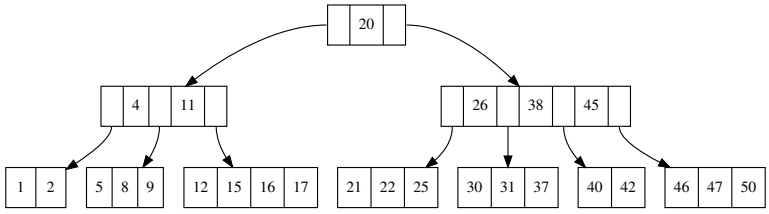
7.2.1 分拆

我们可以通过对节点分拆解决插入时的平衡问题。有两种分拆方法：一种是插入前预先将可能超限的节点进行分拆；另一种是先插入后再通过分拆节点修复平衡。

[1]实际上，元素只需支持小于比较和等于比较。参见本章练习题。



(a) 将 22 插入 2-3-4 树：$22 > 20$，插入右子树；$22 < 26$，插入第一个子节点。



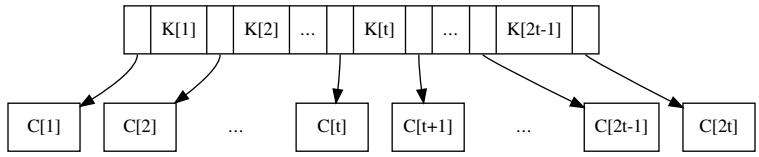
(b) $21 < 22 < 25$，且叶子节点未满。

图 7.3: B 树的插入和二叉搜索树相似

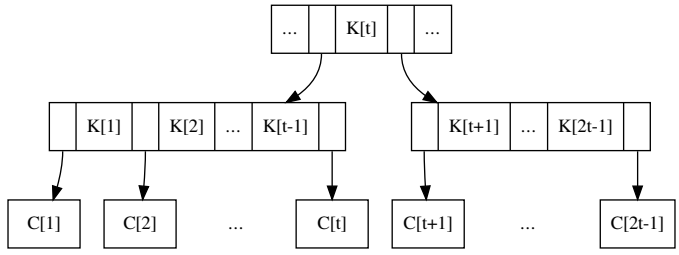
插入前预分拆

如果节点已满，我们可以在插入前预先对节点进行分拆。

一个含有 $t-1$ 个 key 的节点可以按照图7.4所示分拆为 3 个部分。左侧的部分包括前 $t-1$ 个 key 和 t 个子树；右侧的部分包括剩下的 $t-1$ 个 key 和 t 个子树。左右两侧都是合法的 B 树。中间的部分是第 t 个 key。我们可以把它向上推入到父节点中。如果当前节点是根节点，则第 t 个 key 和分拆出的两个较小的子树将组成一个新的根节点。



(a) 分拆前



(b) 分拆后

图 7.4: 分拆节点

给定节点 x，记 $K(x)$ 为节点中所有 key 的列表，$C(x)$ 为全部子树的列表。第 i 个 key 为 $k_i(x)$，第 j 个子树为 $c_j(x)$。下面的算法描述了如何分拆节点 node 中的第 i 个子树：

1: **procedure** SPLIT-CHILD($node, i$)
2: $x \leftarrow c_i(node)$
3: $y \leftarrow$ CREATE-NODE
4: INSERT($K(node), i, k_t(x)$)
5: INSERT($C(node), i+1, y$)
6: $K(y) \leftarrow \{k_{t+1}(x), k_{t+2}(x), ..., k_{2t-1}(x)\}$
7: $K(x) \leftarrow \{k_1(x), k_2(x), ..., k_{t-1}(x)\}$
8: **if** y is not leaf **then**
9: $C(y) \leftarrow \{c_{t+1}(x), c_{t+2}(x), ..., c_{2t}(x)\}$
10: $C(x) \leftarrow \{c_1(x), c_2(x), ..., c_t(x)\}$

下面的 Python 例子程序实现了子树分拆算法。

```
def split_child(node, i):
    t = node.t
    x = node.children[i]
    y = BTree(t)
    node.keys.insert(i, x.keys[t-1])
    node.children.insert(i+1, y)
    y.keys = x.keys[t:]
    x.keys = x.keys[:t-1]
    if not is_leaf(x):
        y.children = x.children[t:]
        x.children = x.children[:t]
```

其中函数 `is_leaf` 判断一个节点是否是叶子节点。

```
def is_leaf(t):
    return t.children == []
```

分拆后，有一个 key 被向上推入到父节点。而父节点有可能已经满了，这样就会违反 B 树的限制条件。

为了解决这一问题，我们可以从根节点开始，沿着插入的路径检查每一个节点。如果路径上的任何节点已经满了，我们就将其分拆。由于我们已经检查过此节点的父节点，因此该父节点所含有的 key 一定少于 $2t-1$。向它推入一个 key 不会破坏 B 树的性质。这一方法只需要自顶向下处理一次而无需任何回溯。

如果根节点需要拆分，就会产生出一个新的根节点，它不含任何 key，此前的根节点成为这个新节点的唯一的子节点。然后我们就可以按照上面的描述进行自顶向下地检查，并最终将新 key 插入。

1: **function** INSERT(T, k)
2: $r \leftarrow T$
3: **if** r is full **then** ▷ 根节点 root 已满

```
        s ← CREATE-NODE
        C(s) ← {r}
        SPLIT-CHILD(s, 1)
        r ← s
    return INSERT-NONFULL(r, k)
```

其中算法 INSERT-NONFULL 假设传入的节点不满而不再做额外的检查。如果传入的节点为叶子节点，就根据待插入 key 的大小将其插入到合适位置；否则，算法就寻找可插入的子节点。如果子节点已满，就进行拆分。

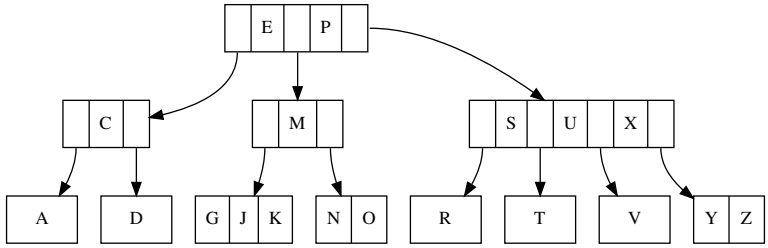
```
function INSERT-NONFULL(T, k)
    if T is leaf then
        i ← 1
        while i ≤ |K(T)| ∧ k > k_i(T) do
            i ← i + 1
        INSERT(K(T), i, k)
    else
        i ← |K(T)|
        while i > 1 ∧ k < k_i(T) do
            i ← i − 1
        if c_i(T) is full then
            SPLIT-CHILD(T, i)
            if k > k_i(T) then
                i ← i + 1
        INSERT-NONFULL(c_i(T), k)
    return T
```

这一算法是递归的。B 树的最小度数 t 通常根据磁盘结构来确定，即使很小的深度也能保存巨大数量的数据（例如 $t = 10$ 的时候，一棵深度为 10 的 B 树可以保存 100 亿数据）。在实现中，递归也可以被消除。这作为一道习题留给读者。

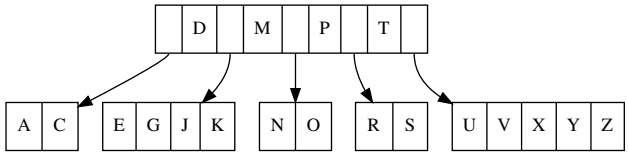
图7.5描述了依次向一个空树插入 G, M, P, X, A, C, D, E, J, K, N, O, R, S, T, U, V, Y, Z 的结果。第一个结果是一棵 2-3-4 树（$t = 2$），第二个结果中的最小度数 $t = 3$。我们可以看出两棵 B 树的异同。

下面的 Python 例子程序实现了这一算法。

```
def insert(tr, key):
    root = tr
    if is_full(root):
        s = BTree(root.t)
        s.children.insert(0, root)
        split_child(s, 0)
        root = s
    return insert_nonfull(root, key)
```



(a) 2-3-4 tree.



(b) $t = 3$

图 7.5: 插入结果

其中向未满节点插入元素的实现如下所示:

```
def insert_nonfull(tr, key):
    if is_leaf(tr):
        ordered_insert(tr.keys, key)
    else:
        i = len(tr.keys)
        while i>0 and key < tr.keys[i-1]:
            i = i-1
        if is_full(tr.children[i]):
            split_child(tr, i)
            if key>tr.keys[i]:
                i = i+1
        insert_nonfull(tr.children[i], key)
    return tr
```

这里,函数 `ordered_insert` 用于将一个元素插入到已序列表中。函数 `is_full` 用以检查一个节点是否含有 $2t - 1$ 个 key。

```
def ordered_insert(lst, x):
    i = len(lst)
    lst.append(x)
    while i>0 and lst[i]<lst[i-1]:
        (lst[i-1], lst[i]) = (lst[i], lst[i-1])
        i=i-1

def is_full(node):
    return len(node.keys) >= 2 * node.t - 1
```

如果容器是用数组实现的, 向末尾添加元素的效率要远高于向中间位置插入的效率。对于长度为 n 的数组, 后者往往是线性时间 $O(n)$ 的。函数 `ordered_insert` 首先将新元素添加到当前容器的末尾, 然后从最后一个元素向前检查相邻两个元素是

否已序。如果大小颠倒，就进行交换操作。

先插入再修复

我们也可以利用和红黑树类似的方法来实现纯函数式的 B 树插入算法。当向红黑树插入时，首先按照普通的二叉搜索树将新 key 插入，然后递归地进行修复以恢复平衡性。B 树可以看作二叉搜索树的扩展，每个节点含有多个 key 和子树。插入时，我们可以暂时不考虑节点是否已满，将新 key 插入后，再进行修复以满足最小度数的限制条件。

$$insert(T, k) = fix(ins(T, k)) \tag{7.5}$$

函数 $ins(T, k)$ 从根节点开始遍历 B 树，找到合适的位置将 k 插入。此后再应用函数 fix 来恢复 B 树的性质。记 B 树为 $T = (K, C, t)$，其中 K 代表全部的 key，C 代表子树，t 代表最小度数。

下面的 Haskell 例子代码定义了 B 树。

```
data BTree a = Node{ keys :: [a]
                   , children :: [BTree a]
                   , degree :: Int} deriving (Eq)
```

根据这一 B 树的定义，我们可以给出如下的 Haskell 插入函数

```
insert tr x = fixRoot $ ins tr x
```

实现函数 $ins(T, k)$ 时，我们要处理两种不同情况：如果 T 是叶子节点，k 就直接被插入到节点中；否则 T 为分支节点，我们需要递归地将 k 插入到某个子节点中。

图7.6给出了分支节点的情况。算法首先定位到插入位置。对于某个 key k_i，若待插入的 key k 满足 $k_{i-1} < k < k_i$，就需要递归将 k 插入到子分支 c_i 中。

待插入位置将节点分成了三个部分：左侧部分、子分支 c_i、和右侧部分。

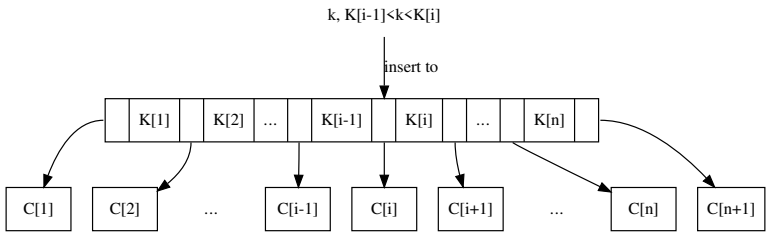
$$ins(T, k) = \begin{cases} (K' \cup \{k\} \cup K'', \phi, t) & : \quad C = \phi, (K', K'') = divide(K, k) \\ make((K', C_1), ins(c, k), (K'', C_2')) & : \quad (C_1, C_2) = split(|K'|, C) \end{cases} \tag{7.6}$$

上式中的第一行处理叶子节点的情况。函数 $divide(K, k)$ 将所有的 key 分成两部分，第一部分中的 key 都不大于 k，第二部分中剩余的 key 都不小于 k:

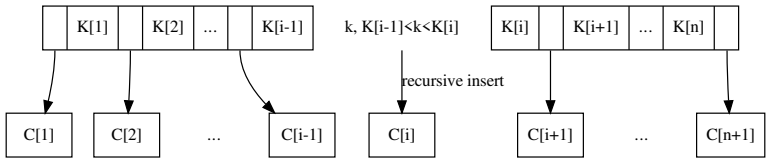
$$K = K' \cup K'' \wedge \forall k' \in K', k'' \in K'' \Rightarrow k' \le k \le k''$$

第二行处理分支节点的情况。函数 $split(n, C)$ 将所有的子树分成 C_1 和 C_2 两部分。其中 C_1 包含了前 n 棵子树；而 C_2 包含剩余的子树。C_2 中的第一棵子树记为 c，其余子树记为 C_2'。

此后，我们需要将 k 递归地插入到子树 c 中。函数 $make$ 接受 3 个参数：其中第一个和第三个分别是一对 key 和子树的列表；第二个参数是一棵子树。它检查用传



(a) 定位到待插入的子分支。



(b) 递归插入。

图 7.6: 向分支节点插入 key

入的 key 和子树构造的 B 树节点是否会违反最小度数限制，如果违反，就进行适当的修复。

$$make((K', C'), c, (K'', C'')) = \begin{cases} fixFull((K', C'), c, (K'', C'')) & : \quad full(c) \\ (K' \cup K'', C' \cup \{c\} \cup C'', t) & : \quad otherwise \end{cases} \tag{7.7}$$

其中函数 $full(c)$ 检查节点 c 是否已满。如果满，函数 $fixFull$ 将节点 c 进行分拆，并且用分拆后推上来的 key 来构造一个新的 B 树节点。

$$fixFull((K', C'), c, (K'', C'')) = (K' \cup \{k'\} \cup K'', C' \cup \{c_1, c_2\} \cup C'', t) \tag{7.8}$$

这里 $(c_1, k', c_2) = split(c)$。在分拆中，前 $t-1$ 个 key 和前 t 个子树被抽出构造一个新节点，后 $t-1$ 个 key 和后 t 个子树被用于构造另一个新节点；第 t 个 key k' 被向上推入到 key 中。

使用上述定义的函数，我们可以最终实现 $fix(T)$ 以完成函数式的 B 树插入算法。它首先检查根节点是否含有过多的 key，如果超过限制，就进行分拆。分拆的结果被用于构造一个新节点，因此树的高度会增加 1。

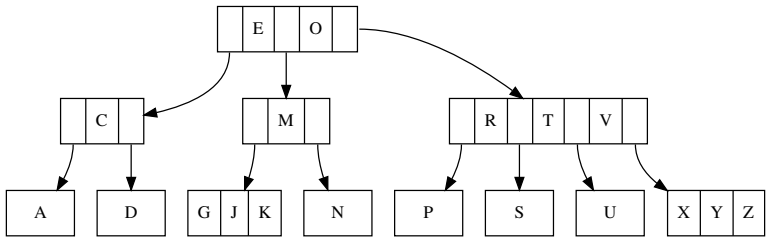
$$fix(T) = \begin{cases} c & : \quad T = (\phi, \{c\}, t) \\ (\{k'\}, \{c_1, c_2\}, t) & : \quad full(T), (c_1, k', c_2) = split(T) \\ T & : \quad otherwise \end{cases} \tag{7.9}$$

下面的 Haskell 例子程序实现了 B 树的插入算法。

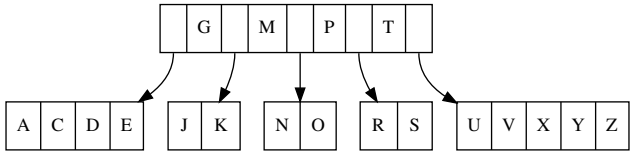
```
import qualified Data.List as L

ins (Node ks [] t) x = Node (L.insert x ks) [] t
ins (Node ks cs t) x = make (ks', cs') (ins c x) (ks'', cs'')
```

```
    where
      (ks', ks'') = L.partition (<x) ks
      (cs', (c:cs'')) = L.splitAt (length ks') cs

fixRoot (Node [] [tr] _) = tr -- shrink height
fixRoot tr = if full tr then Node [k] [c1, c2] (degree tr)
             else tr
    where
      (c1, k, c2) = split tr

make (ks', cs') c (ks'', cs'')
    | full c = fixFull (ks', cs') c (ks'', cs'')
    | otherwise = Node (ks'++ks'') (cs'++[c]++cs'') (degree c)

fixFull (ks', cs') c (ks'', cs'') = Node (ks'++[k]++ks'')
                                         (cs'++[c1,c2]++cs'') (degree c)
    where
      (c1, k, c2) = split c

full tr = (length $ keys tr) > 2*(degree tr)-1
```

图7.7给出了不断向空树中插入“GMPXACDEJKNORSTUVYZ”的两个不同结果。



(a) 2-3-4 树的插入结果。



(b) 最小度数 $t = 3$ 的 B 树插入结果。

图 7.7: 先插入再修复的结果

和图7.5所示的命令式的插入结果相比较，我们可以看到它们的不同之处。它们都是满足 B 树性质的合法结果。

7.3 删除

最小度数为 t 的 B 树中，除根节点外，任何节点中的 key 都不能少于 $t-1$ 个。从节点中删除一个 key 后，有可能会违反这一平衡性质。

同插入操作一样，我们可以采用类似的策略：或者在删除前进行额外的准备工作，以保证节点含有足够多的 key；或者在删除后对节点进行修复，以避免含有的 key 过少。

7.3.1 删除前预合并

我们先处理最简单的情况：如果待删除的 key k 所在的节点为一叶子节点 x，我们可以直接将 k 从 x 中删除。如果 x 是根节点（树中的唯一节点），我们无需担心删除后含有的 key 过少。以上两种，我们称之为情况 1。

通常情况下，我们从根节点开始，自顶向下沿着一条路径定位到 k 所在的节点 x。如果 x 是一分支节点，则有如下三种子情况：

- 子情况 2a：如果 k 前面的子节点 y 含有足够多的 key（多于 t），我们用 y 中 k 的前驱元素 k' 替换掉 x 中的 k，然后递归地在 y 中将 k' 删除。

  其中，k 的前驱元素就是子节点 y 中的最后一个 key。

  图7.8描述了这种情况。

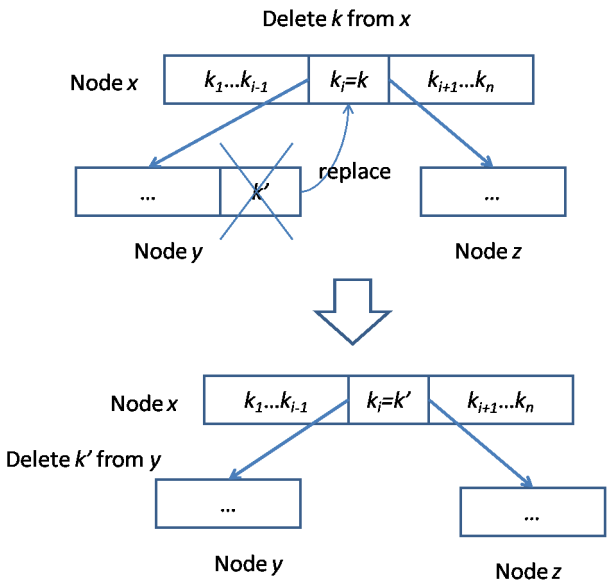


图 7.8: 使用前驱元素替换并递归进行删除

- 子情况 2b：如果 y 含有的 key 不足，但是 k 的后继子节点 z 含有的 key 多于 t，我们可以将 x 中的元素 k 用 z 中 k 的后继元素 k'' 来替换，然后再递归地将 z 中的 k'' 删除。

  其中，k 的后继元素就是子节点 z 中的第一个 key。

  图7.9描述了这种情况。

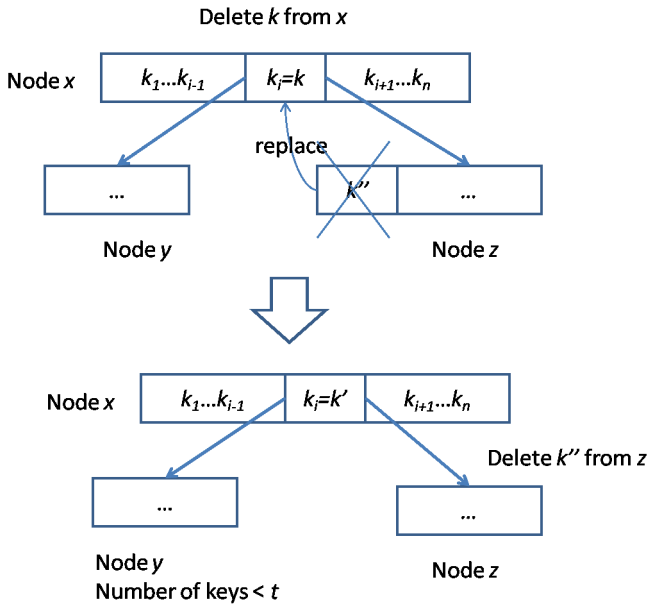


图 7.9: 使用后继元素替换并递归进行删除

- 子情况 2c：否则，如果 y 和 z 含有的 key 都不足，我们可以将 y、k 和 z 合并成一个新节点，它恰好含有 $2t-1$ 个 key，此后，我们就可以针对这个新节点进行递归删除。

  这里有一个特殊情况：如果合并后的节点不含有任何 key，也就是说，k 是 x 中的唯一 key，而 y 和 z 是 x 仅有的两个子节点。这时我们需要将树的高度降低一层。

图7.10描述了这种情况。

还有一种情况，如果 k 不是节点 x 中的 key，我们需要在 x 中找到一个子节点 c_i，使得 k 在子树 c_i 中。在对 c_i 进行递归删除前，我们需要预先确定 c_i 至少含有 t 个 key。如果含有的 key 不足，就需要进行如下的调整。

- 子情况 3a：我们检查 c_i 的前后兄弟节点 c_{i-1} 和 c_{i+1}。如果任何一个节点包含有足够的 key（至少 t 个），我们就将 x 中的一个 key 向下移动到 c_i 中，并将含有足够多 key 的兄弟节点中的一个 key 向上移动到 x 中。同时，我们还需要将兄弟节点中相应的子节点移动到 c_i 中。

  这一操作使得 c_i 含有足够多的 key 以便进行后面的删除。接下来我们可以从 c_i 中递归删除 k。

  图7.11描述了这一情况。

- 子情况 3b：如果左右两个兄弟节点含有的 key 都不足，我们可以将 c_i，x 中的一个 key，和任一兄弟节点合并为一个新节点。然后针对这一节点执行删除操作。

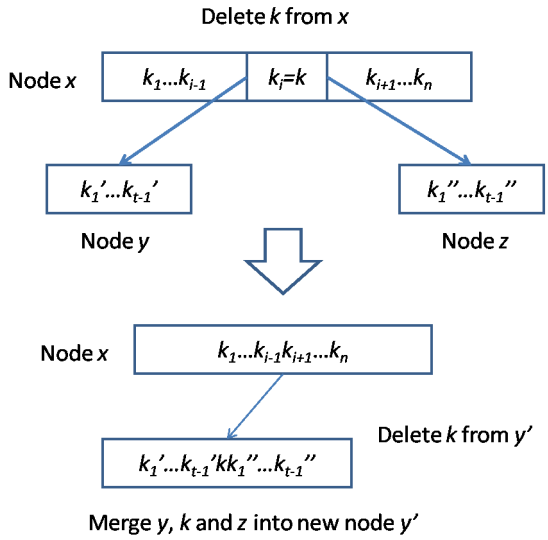


图 7.10: 合并后再删除

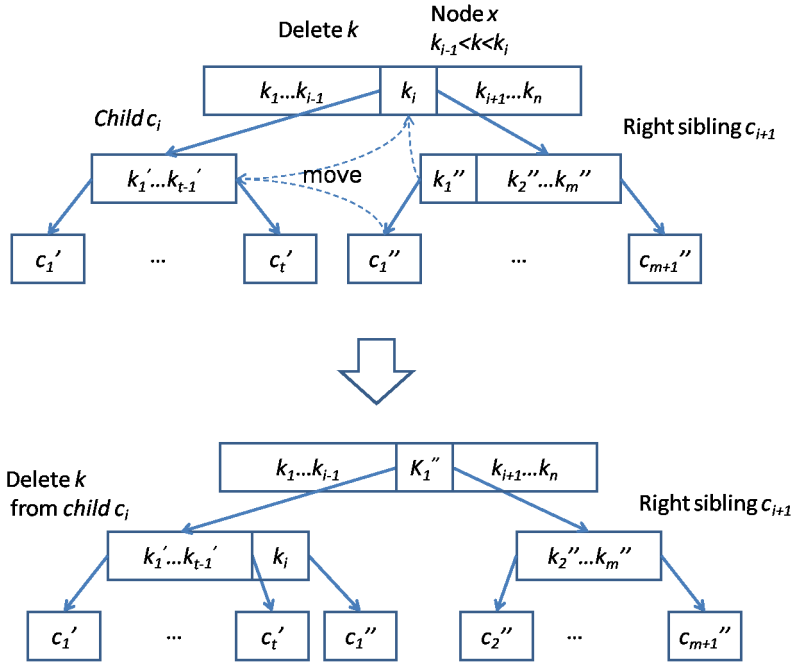


图 7.11: 向右侧的兄弟节点“借”一个 key

图7.12描述了这一情况。

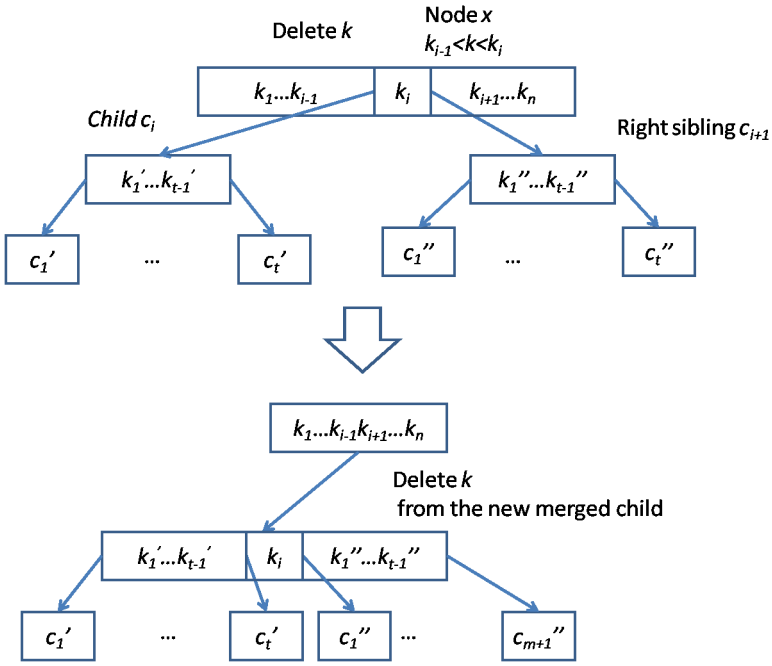


图 7.12: 将 c_i、k 和 c_{i+1} 合并为一个新节点

为了实现删除算法，我们需要先定义一些辅助函数。函数 CAN-DEL 检查一个节点是否含有足够多的 key 以执行删除操作。

```
1: function CAN-DEL(T)
2:     return |K(T)| ≥ t
```

过程 MERGE-CHILDREN(T, i) 将子节点 $c_i(T)$、key $k_i(T)$ 和子节点 $c_{i+1}(T)$ 合并成一个新节点。

```
1: procedure MERGE-CHILDREN(T, i)          ▷ 将 c_i(T)、k_i(T) 和 c_{i+1}(T) 合并
2:     x ← c_i(T)
3:     y ← c_{i+1}(T)
4:     K(x) ← K(x) ∪ {k_i(T)} ∪ K(y)
5:     C(x) ← C(x) ∪ C(y)
6:     REMOVE-AT(K(T), i)
7:     REMOVE-AT(C(T), i + 1)
```

这一过程从给定的树 T 中定位到第 i 个子节点和 key，将它们和第 $i+1$ 个节点合并，然后从 T 中将第 i 个 key 和第 $i+1$ 个子节点删除。

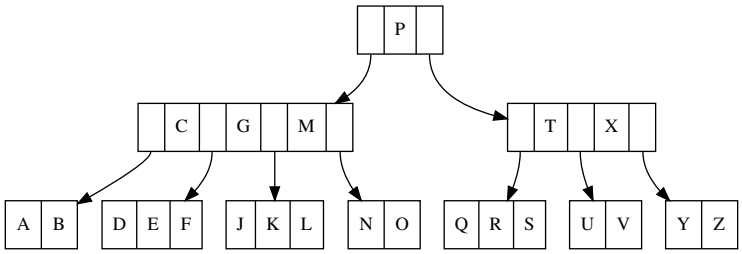
使用上述函数，我们可以分别处理三种不同的情况，从而定义下面的 B 树删除算法。

```
1: function DELETE(T, k)
```

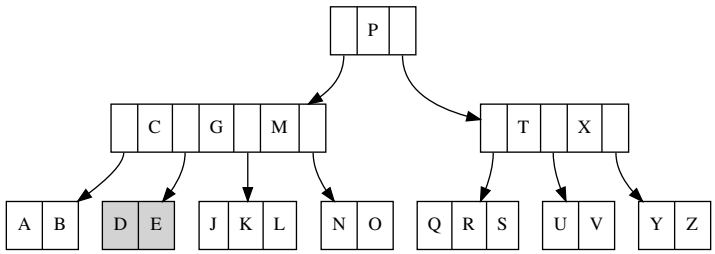
```
    i ← 1
    while i ≤ |K(T)| do
        if k = k_i(T) then
            if T is leaf then                                   ▷ 情况 1
                REMOVE(K(T), k)
            else                                                ▷ 情况 2
                if CAN-DEL(c_i(T)) then                         ▷ 情况 2a
                    k_i(T) ← LAST-KEY(c_i(T))
                    DELETE(c_i(T), k_i(T))
                else if CAN-DEL(c_{i+1}(T)) then                ▷ 情况 2b
                    k_i(T) ← FIRST-KEY(c_{i+1}(T))
                    DELETE(c_{i+1}(T), k_i(T))
                else                                            ▷ 情况 2c
                    MERGE-CHILDREN(T, i)
                    DELETE(c_i(T), k)
                    if K(T) = NIL then
                        T ← c_i(T)                              ▷ 缩小高度
            return T
        else if k < k_i(T) then
            Break
        else
            i ← i + 1

    if T is leaf then
        return T                                                ▷ k 不在 T 中
    if ¬ CAN-DEL(c_i(T)) then                                   ▷ 情况 3
        if i > 1∧ CAN-DEL(c_{i−1}(T)) then                      ▷ 情况 3a：左侧兄弟
            INSERT(K(c_i(T)), k_{i−1}(T))
            k_{i−1}(T) ← POP-BACK(K(c_{i−1}(T)))
            if c_i(T) isn't leaf then
                c ← POP-BACK(C(c_{i−1}(T)))
                INSERT(C(c_i(T)), c)
        else if i ≤ |C(T)|∧ CAN-DEL(c_{i_1}(T)) then            ▷ 情况 3a：右侧兄弟
            APPEND(K(c_i(T)), k_i(T))
            k_i(T) ← POP-FRONT(K(c_{i+1}(T)))
            if c_i(T) isn't leaf then
                c ← POP-FRONT(C(c_{i+1}(T)))
                APPEND(C(c_i(T)), c)
```

```
39:         else                                                    ▷ 情况 3b
40:             if i > 1 then
41:                 MERGE-CHILDREN(T, i − 1)
42:             else
43:                 MERGE-CHILDREN(T, i)
44:     DELETE(c_i(T), k)                                           ▷ 递归删除
45:     if K(T) = NIL then                                          ▷ 缩小高度
46:         T ← c_1(T)
47:     return T
```

图7.13、7.14和7.15描述了删除中的各个步骤，被改变的节点用灰色显示。



(a) 删除前的 B 树

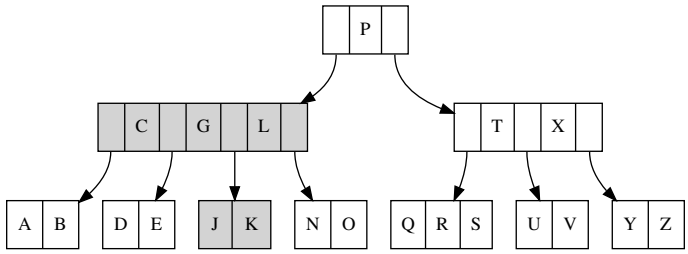


(b) 删除 key ‘F’，情况 1

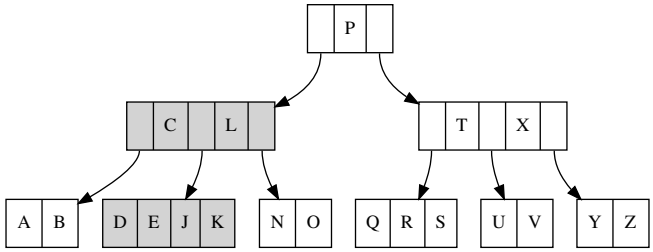
图 7.13: B 树删除的结果（1）

下面的 Python 例子程序实现了 B 树的删除算法。

```
def can_remove(tr):
    return len(tr.keys) ≥ tr.t

def replace_key(tr, i, k):
    tr.keys[i] = k
    return k

def merge_children(tr, i):
    tr.children[i].keys += [tr.keys[i]] + tr.children[i+1].keys
    tr.children[i].children += tr.children[i+1].children
    tr.keys.pop(i)
    tr.children.pop(i+1)
```

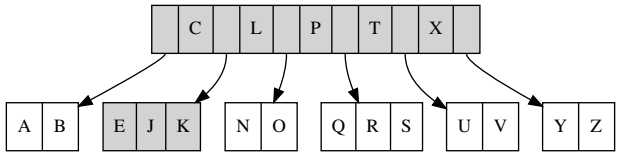


(a) 删除 key ‘M’后，子情况 2a

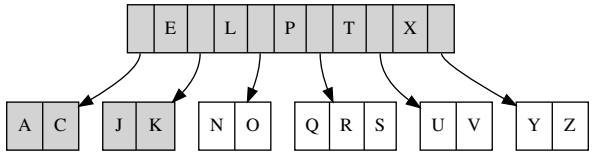


(b) 删除 key ‘G’后，子情况 2c

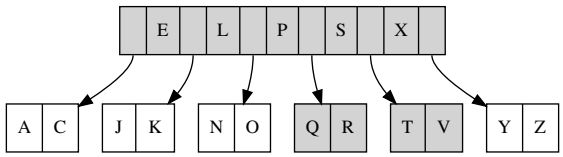
图 7.14: B 树删除的结果（2）



(a) 删除 key ‘D’后，子情况 3b，树的高度减少 1



(b) 删除 key ‘B’后，子情况 3a，向右侧兄弟节点”借“一个 key



(c) 删除 key ‘U’后，子情况 3a，向左侧兄弟节点“借”一个 key

图 7.15: B 树删除的结果（3）

```
def B_tree_delete(tr, key):
    i = len(tr.keys)
    while i>0:
        if key == tr.keys[i-1]:
            if tr.leaf: #情况 1
                tr.keys.remove(key)
            else:#情况 2
                if tr.children[i-1].can_remove():#情况 2a
                    key = tr.replace_key(i-1, tr.children[i-1].keys[-1])
                    B_tree_delete(tr.children[i-1], key)
                elif tr.children[i].can_remove():#情况 2b
                    key = tr.replace_key(i-1, tr.children[i].keys[0])
                    B_tree_delete(tr.children[i], key)
                else:#情况 2c
                    tr.merge_children(i-1)
                    B_tree_delete(tr.children[i-1], key)
                    if tr.keys==[]:#缩减树的高度
                        tr = tr.children[i-1]
            return tr
        elif key > tr.keys[i-1]:
            break
        else:
            i = i-1
    #情况 3
    if tr.leaf:
        return tr #key 不存在
    if not tr.children[i].can_remove():
        #情况 3a
        if i>0 and tr.children[i-1].can_remove():#左侧兄弟
            tr.children[i].keys.insert(0, tr.keys[i-1])
            tr.keys[i-1] = tr.children[i-1].keys.pop()
            if not tr.children[i].leaf:
                tr.children[i].children.insert(0, tr.children[i-1].children.pop())
        elif i<len(tr.children) and tr.children[i+1].can_remove():#右侧兄弟
            tr.children[i].keys.append(tr.keys[i])
            tr.keys[i]=tr.children[i+1].keys.pop(0)
            if not tr.children[i].leaf:
                tr.children[i].children.append(tr.children[i+1].children.pop(0))
        else:#情况 3b
            if i>0:
                tr.merge_children(i-1)
            else:
                tr.merge_children(i)
    B_tree_delete(tr.children[i], key)
    if tr.keys==[]:#缩减树的高度
        tr = tr.children[0]
    return tr
```

7.3.2 先删除再修复

删除前预合并算法比较复杂了，需要处理不同的情况，每种情况又含有若干子情况。

我们也可以换一种思路来设计删除算法。即先删除，然后再进行必要的修复。这种策略和先插入再修复相类似。

$$delete(T, k) = fix(del(T, k)) \tag{7.10}$$

从 B 树删除一个 key 时，我们先从根节点开始，自顶向下定位到这个 key 所在的节点。

如果这一节点是一个叶子节点，我们就删掉相应的 key，然后检查节点种剩余的 key 是否太少以至于无法满足 B 树的平衡条件。

如果这一节点是一个分支节点，删掉 key 后它就被分为两部分。我们合并它们。这一合并过程是递归的，如图7.16所示。

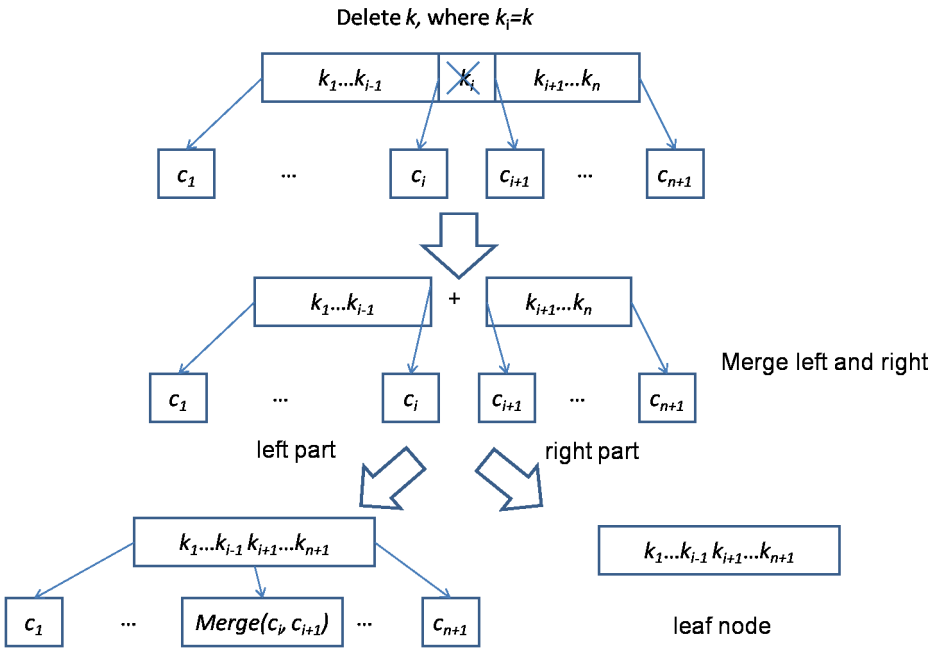


图 7.16: 从分支节点种删除 key。删除 k_i 后节点分成了两部分。递归将它们合并直到这两部分都是叶子节点。

合并时，如果待合并的两个节点不是叶子节点，我们将 key 合到一起，然后递归地将左侧部分的最后一个子树和右侧部分的第一个子树合并。否则，如果它们都是叶子节点，我们只需要将 key 合并到一起。

到目前为止，待删除的 key 已经从树中去掉了。但是由此导致节点中 key 的减少可能会违反 B 树的平衡条件。我们需要从根节点开始，沿着删除时经过的路径进行修复。

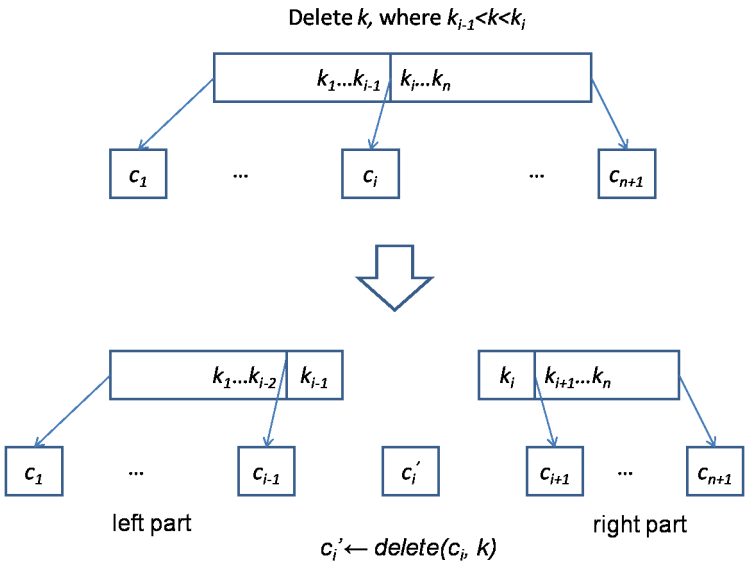


图 7.17: 从节点 c_i 中删除 key k 后，结果为 c_i'。修复时，我们用左侧部分、c_i' 和右侧部分构造一个新节点。

经过递归删除，路径上的任何一个分支节点都被分成了三部分：左侧部分包含了所有不大于 k 的 key，包括 $k_1, k_2, ..., k_{i-1}$ 和子树 $c_1, c_2, ..., c_{i-1}$；右侧部分包含了全部不小于 k 的 key，包括 $k_i, k_{i+1}, ..., k_{n+1}$ 和子树 $c_{i+1}, c_{i+2}, ..., c_{n+1}$；$k$ 被递归地从子树 c_i 中删除，我们将结果记为 c_i'。如图7.17所示。

此时，我们需要检查 c_i' 是否包含了足够多的 key。如果不足（少于 $t-1$ 个，注意这里和删除前预合并不同，后者判断是否少于 t 个），我们可以从左侧，或者右侧“借”来一对 key 和子树，这是分拆操作的逆向操作。图7.18描述了向左侧“借”一对 key 和子树的情形。

如果左侧和右侧都为空，我们只需要把 c_i' 推向上层。

记 B 树为 $T = (K, C, t)$，其中 K 和 C 分别为 key 和子树。函数 $del(T, k)$ 从树中删除 key k。

$$del(T,k) = \begin{cases} (delete(K,k), \phi, t) & : \quad C = \phi \\ merge((K_1, C_1, t), (K_2, C_2, t)) & : \quad k_i = k \\ make((K_1', C_1'), del(c,k), (K_2', C_2')) & : \quad k \notin K \end{cases} \tag{7.11}$$

如果没有任何子树 $C = \phi$，说明 T 是一叶子节点。我们直接将 k 删除。否则，T 为一个内部分支节点。如果 $k \in K$，将 k 删除后所有的 key 和子树被分成了两部分：(K_1, C_1) 和 (K_2, C_2)。它们接下来被递归地合并。

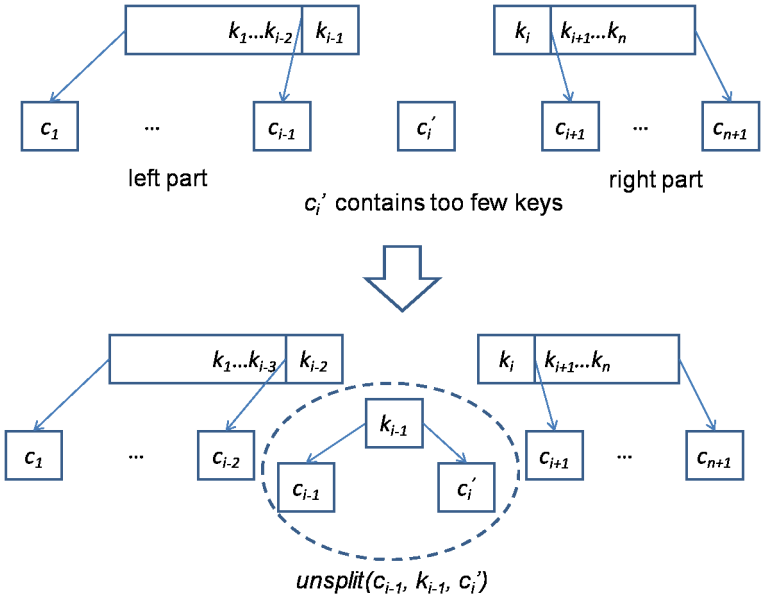


图 7.18: 向左侧部分“借”一对 key 和子树，然后进行分拆的逆操作

$$
\begin{aligned}
K_1 &= \{k_1, k_2, ..., k_{i-1}\} \\
K_2 &= \{k_{i+1}, k_{i+2}, ..., k_m\} \\
C_1 &= \{c_1, c_2, ..., c_i\} \\
C_2 &= \{c_{i+1}, c_{i+2}, ..., c_{m+1}\}
\end{aligned}
$$

如果 $k \notin K$，我们需要定位到一个子树 c，然后递归地从这个子树中删除 k。

$$
\begin{aligned}
(K_1', K_2') &= (\{k' | k' \in K, k' < k\}, \{k' | k' \in K, k < k'\}) \\
(C_1', \{c\} \cup C_2') &= splitAt(|K_1'|, C)
\end{aligned}
$$

递归合并函数被定义如下：当合并两棵树 $T_1 = (K_1, C_1, t)$ 和 $T_2 = (K_2, C_2, t)$ 时，如果它们都是叶子节点，我们将两组 key 连接到一起形成一个新的叶子节点。否则，我们将 C_1 中的最后一棵子树和 C_2 中的第一棵子树递归合并。然后调用 $make$ 函数构造一棵新树。若 C_1 和 C_2 不为空，记 C_1 中的最后一棵子树为 $c_{1,m}$，其余子树为 C_1'；记 C_2 中的第一棵子树为 $C_{2,1}$，其余子树为 C_2'。下面的公式定义了合并函数：

$$
merge(T_1, T_2) = \begin{cases}
(K_1 \cup K_2, \phi, t) & : \quad C_1 = C_2 = \phi \\
make((K_1, C_1'), merge(c_{1,m}, c_{2,1}), (K_2, C_2')) & : \quad otherwise
\end{cases} \tag{7.12}
$$

我们此前定义的 $make$ 函数仅仅处理了由于插入造成节点中含有过多 key 的情况。我们可以对它进行修改，使得它能够处理由于删除造成 key 过少的情况。

$$
make((K', C'), c, (K'', C'')) = \begin{cases}
fixFull((K', C'), c, (K'', C'')) & : \quad full(c) \\
fixLow((K', C'), c, (K'', C'')) & : \quad low(c) \\
(K' \cup K'', C' \cup \{c\} \cup C'', t) & : \quad otherwise
\end{cases} \tag{7.13}
$$

其中 $low(T)$ 检查节点 T 含有的 key 是否少于 $t-1$。函数 $fixLow(P_l, c, P_r)$ 接受三个参数：左侧的 key 和子树对 P_l、一个子节点 c、以及右侧的 key 和子树对 P_r。如果左侧部分不为空，我们就从左侧“借”一对 key 和子树，然后进行逆分拆操作使得节点含有足够多的 key，然后递归地调用 $make$。否则，如果右侧不为空，我们就向右侧“借”一对 key 和子树。如果左右都为空，我们就将子节点直接返回作为结果。这种情况下，树的高度会减低。

令左侧部分为 $P_l = (K_l, C_l)$，如果 K_l 不为空，记最后一对 key 和子树分别为 $k_{l,m}$ 和 $c_{l,m}$。剩余的 key 和子树记为 K_l' and C_l'。同样，令右侧部分为 $P_r = (K_r, C_r)$，如果 K_r 不为空，记第一对 key 和子树分别为 $k_{r,1}$ 和 $c_{r,1}$。剩余的 key 和子树记为 K_r' and C_r'。函数 $fixLow$ 定义如下：

$$fixLow(P_l, c, P_r) = \begin{cases} make((K_l', C_l'), unsplit(c_{l,m}, k_{l,m}, c), (K_r, C_r)) & : \quad K_l \neq \phi \\ make((K_r, C_r), unsplit(c, k_{r,1}, c_{r,1}), (K_r', C_r')) & : \quad K_r \neq \phi \\ c & : \quad otherwise \end{cases} \tag{7.14}$$

函数 $unsplit(T_1, k, T_2)$ 是分拆的逆操作，它用两个子树和一个 key 构造一棵新的 B 树。

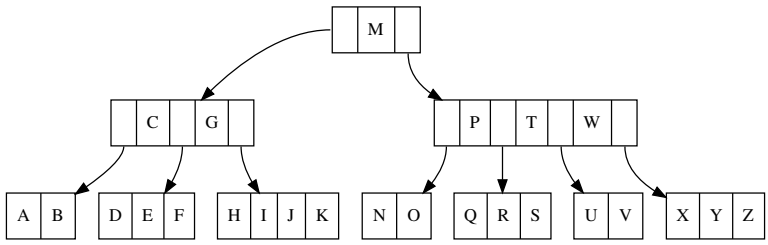
$$unsplit(T_1, k, T_2) = (K_1 \cup \{k\} \cup K_2, C_1 \cup C_2, t) \tag{7.15}$$

下面的 Haskell 例子程序实现了 B 树的删除算法。

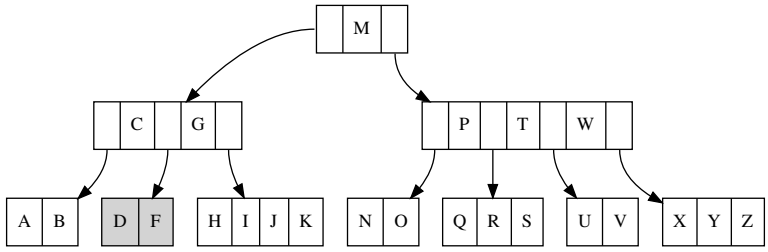
```
import qualified Data.List as L

delete tr x = fixRoot $ del tr x

del:: (Ord a) ⇒ BTree a → a → BTree a
del (Node ks [] t) x = Node (L.delete x ks) [] t
del (Node ks cs t) x =
    case L.elemIndex x ks of
      Just i → merge (Node (take i ks) (take (i+1) cs) t)
                     (Node (drop (i+1) ks) (drop (i+1) cs) t)
      Nothing → make (ks', cs') (del c x) (ks'', cs'')
    where
      (ks', ks'') = L.partition (<x) ks
      (cs', (c:cs'')) = L.splitAt (length ks') cs

merge (Node ks [] t) (Node ks' [] _) = Node (ks++ks') [] t
merge (Node ks cs t) (Node ks' cs' _) = make (ks, init cs)
                                            (merge (last cs) (head cs'))
                                            (ks', tail cs')

make (ks', cs') c (ks'', cs'')
    | full c = fixFull (ks', cs') c (ks'', cs'')
    | low c  = fixLow  (ks', cs') c (ks'', cs'')
    | otherwise = Node (ks'++ks'') (cs'++[c]++cs'') (degree c)

low tr = (length $ keys tr) < (degree tr)-1
```

```
fixLow (ks'@(_:_), cs') c (ks'', cs'') = make (init ks', init cs')
                                             (unsplit (last cs') (last ks') c)
                                             (ks'', cs'')
fixLow (ks', cs') c (ks''@(_:_), cs'') = make (ks', cs')
                                             (unsplit c (head ks'') (head cs''))
                                             (tail ks'', tail cs'')
fixLow _ c _ = c

unsplit c1 k c2 = Node ((keys c1)++[k]++(keys c2))
                       ((children c1)++(children c2)) (degree c1)
```

使用先删除再修复的方法从同样的 B 树中依次删除同样的 key，得到的结果和删除前预合并的有所不同，如图7.19、7.20和7.21。但是它们都是满足平衡条件的合法的 B 树。



(a) 删除前的 B 树



(b) 删除 key ‘E’ 后

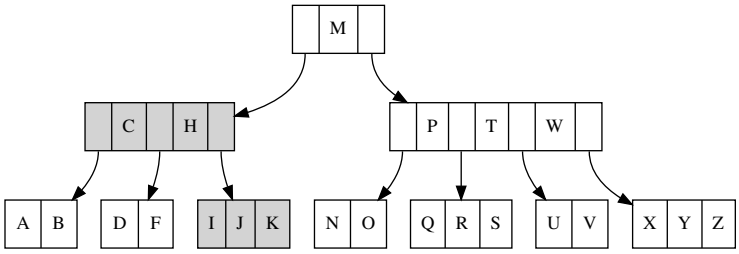
图 7.19: 先删除再修复的结果（1）

7.4 搜索

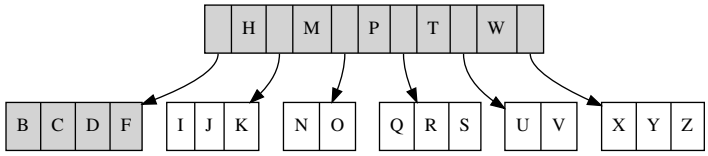
我们可以将二叉搜索树的搜索算法进行抽象概括，从而获得 B 树的搜索算法。

对于二叉树，每次只有两种选择，或者向左，或者向右。但是 B 树中存在多个选择。

1: **function** SEARCH(T, k)
2: **loop**
3: $i \leftarrow 1$

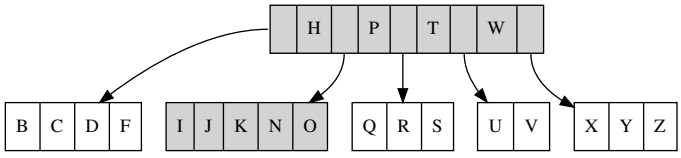


(a) 删除 key ‘G’后

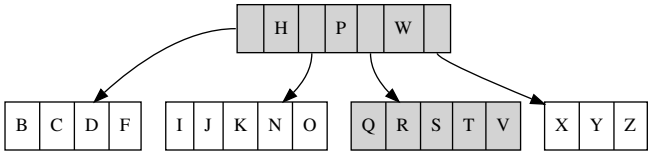


(b) 删除 key ‘A’后

图 7.20: 先删除再修复的结果（2）



(a) 删除 key ‘M’后



(b) 删除 key ‘U’后

图 7.21: 先删除再修复的结果（3）

```
    while i ≤ |K(T)| ∧ k > k_i(T) do
        i ← i + 1
    if i ≤ |K(T)| ∧ k = k_i(T) then
        return (T, i)
    if T is leaf then
        return NIL                                  ▷ k 不存在
    else
        T ← c_i(T)
```

从根节点开始,算法按照从小到大的顺序逐一检查每个 key。如果发现匹配的 key,就返回当前节点和 key 的索引作为结果。否则,如果找到某一位置 i 使得 $k_i < k < k_{i+1}$,接下来就在子树 c_{i+1} 中搜索这一 key。如果到达了某一叶子节点还没有找到 key,就返回一个空值表示不存在。

下面的 Python 例子程序实现了 B 树搜索算法。

```
def B_tree_search(tr, key):
    while True:
        for i in range(len(tr.keys)):
            if key ≤ tr.keys[i]:
                break
        if key == tr.keys[i]:
            return (tr, i)
        if tr.leaf:
            return None
        else:
            if key > tr.keys[-1]:
                i=i+1
            tr = tr.children[i]
```

也可以用递归的方式实现 B 树搜索算法。当在 B 树 $T = (K, C, t)$ 中搜索 key k 时,我们首先使用 k 将所有的 key 分成两部分。

$$K_1 = \{k' | k' < k\}$$
$$K_2 = \{k' | k \leq k'\}$$

即 K_1 包含所有小于 k 的 key;而 K_2 包含其余的部分。如果 K_2 的第一个元素恰好等于 k,则搜索成功,否则我们需要递归地在子树 $c_{|K_1|+1}$ 中搜索。

$$search(T, k) = \begin{cases} (T, |K_1| + 1) & : & k \in K_2 \\ \phi & : & C = \phi \\ search(c_{|K_1|+1}, k) & : & otherwise \end{cases} \tag{7.16}$$

下面的 Haskell 例子程序实现了这一算法。

```
search :: (Ord a)⇒ BTree a → a → Maybe (BTree a, Int)
search tr@(Node ks cs _) k
    | matchFirst k $ drop len ks = Just (tr, len)
```

```
  | otherwise = if null cs then Nothing
                else search (cs !! len) k
  where
    matchFirst x (y:_) = x==y
    matchFirst x _ = False
    len = length $ filter (<k) ks
```

7.5 小结

本章中，我们介绍了 B 树，它是二叉搜索树的一种扩展。我们跳过了有关磁盘访问的背景知识，读者可以参考 [4] 加以了解。我们给出了主要三种操作：插入、删除和查找的命令式和函数式算法。它们都从根节点向叶子节点进行查找，算法执行的时间和树的高度成正比。由于 B 树总是平衡的，因此对于含有 n 个元素的 B 树，这些操作的时间就可以保证是 $O(\lg n)$ 的。

练习 7.1

- 在插入时，我们需要找到合适的位置，使得左侧的 key 都小于待插入的元素，而右侧的 key 都大于它。实际上，元素只要支持小于和等于比较就可以了。请改动插入算法，放松这一限制。
- 我们假设 B 树中不存在待插入的元素。修改算法使得重复的元素保存在一链表中。
- 修改命令式 B 树算法，去除其中的递归调用。

第八章　二叉堆

8.1　简介

堆是被广泛应用的一种数据结构。堆可以用于解决很多实际问题，包括排序、带有优先级的调度，实现图算法等等 [40]。

堆有很多不同的实现，其中最常见的一种通过数组来表示二叉树 [4]，进而实现堆。例如 C++ 标准库 STL 中的 heap 和 Python 库中的 heapq 都是这样实现堆的。由 R.W. Floyd 给出的最高效的堆排序算法也是利用这个实现 [41][42]。

堆是一种通用的概念，它也可以由数组以外的其他数据结构来实现。本章中，我们给出一些使用二叉树来实现的堆，包括左偏堆（Leftist Heap）、skew 堆（也有文献译为“斜堆”）、和伸展堆（splay heap）。它们非常适合纯函数式实现 [3]。

堆是一种满足如下性质的数据结构：

- 顶部（top）总是保存着最小（或最大）的元素；
- 弹出（pop）操作将顶部元素移除，同时保持堆的性质，新的顶部元素仍然是剩余元素中的最小（或最大）值；
- 将新元素插入到堆中仍然保持堆的性质，顶部元素还是所有元素中的最小（或最大）值；
- 其他操作（例如将两个堆合并），都会保持堆的性质。

这一定义是递归的。它并没有限定实现堆的低层数据结构。

我们称顶部保存最小元素的堆为最小堆，顶部保存最大元素的堆为最大堆。

8.2　用数组实现隐式二叉堆

考虑堆的定义，我们可以用树来实现堆。一种直观的想法是将最小（或最大）元素保存在树的根节点。获取“顶部”元素时，我们可以直接返回根节点中的数据。执行“弹出”操作时，我们将根节点删除，然后从子节点中重建树。

我们称使用二叉树实现的堆为二叉堆。本章介绍三种不同的二叉堆实现。

8.2.1 定义

第一种实现称为隐式二叉树。考虑如何用数组来表示一棵完全二叉树（例如，有些编程语言中没有结构或记录等复合数据类型，只能使用数组来定义二叉树）。我们可以将全部元素自顶向下（从根节点开始，到叶子节点为止）压缩放入数组中。

图8.1展示了一棵完全二叉树和它相应的数组表示形式。

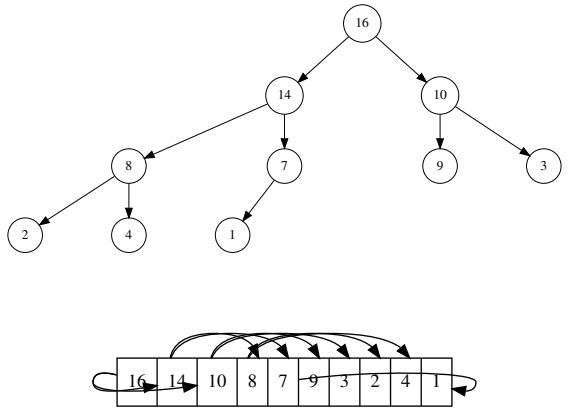


图 8.1: 完全二叉树到数组的映射

树和数组之间的映射可以定义如下（令数组的索引从 1 开始）：

1: **function** PARENT(i)
2: **return** $\lfloor \frac{i}{2} \rfloor$

3: **function** LEFT(i)
4: **return** $2i$

5: **function** RIGHT(i)
6: **return** $2i + 1$

对数组中第 i 个元素代表的节点，由于二叉树是完全的，我们可以通过定位到第 $\lfloor i/2 \rfloor$ 个元素找到它的父节点；它的左子树对应第 $2i$ 个元素，而右子树对应第 $2i + 1$ 个元素。如果子节点的索引超出了数组的长度，说明它不含有相应的子树（例如叶子节点）。

在实际的应用中，父节点和子树的访问可以通过位运算实现，例如下面的 C 代码。注意，代码中的索引从 0 开始。

```
#define PARENT(i) ((((i) + 1) >> 1) - 1)

#define LEFT(i) (((i) << 1) + 1)

#define RIGHT(i) (((i) + 1) << 1)
```

8.2.2 Heapify

堆算法中最重要的部分就是维护堆的性质：即顶部元素为最小（或最大）元素。

对于用数组表示的二叉堆，给定任何索引为 i 的节点，我们可以检查它的两个子节点是否都不小于父节点。如果不满足，我们可以通过不断交换、检查，使得父节点保存最小值 [4]。注意：这里我们假设 i 的两棵子树都是合法的堆。

下面的算法从给定的数组索引开始，迭代检查所有的子节点以保持最小堆性质。

```
1:  function HEAPIFY(A, i)
2:      n ← |A|
3:      loop
4:          l ← LEFT(i)
5:          r ← RIGHT(i)
6:          smallest ← i
7:          if l < n ∧ A[l] < A[i] then
8:              smallest ← l
9:          if r < n ∧ A[r] < A[smallest] then
10:             smallest ← r
11:         if smallest ≠ i then
12:             EXCHANGE A[i] ↔ A[smallest]
13:             i ← smallest
14:         else
15:             return
```

算法接受一个数组 A 和一个索引 i，$A[i]$ 的两个子节点都不应比它小。否则，我选出最小的元素保存在 $A[i]$，并将较大的元素交换至子树，然后算法自顶向下检查并修复堆的性质直到叶子节点或者没有发现任何违反堆性质的情况。

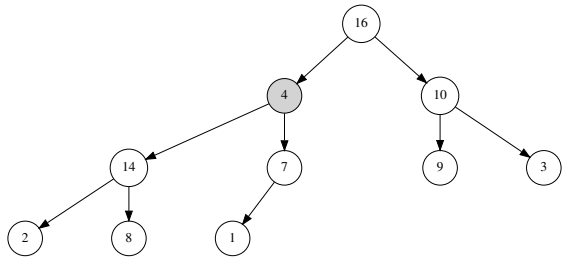
HEAPIFY 的时间复杂度为 $O(\lg n)$，其中 n 是元素的总数。这是因为上述算法中的循环次数和完全二叉树的高度成正比。

在具体的实现中，元素之间的比较运算可以用参数的形式传入，这样同一实现就即可以支持最小堆，也支持最大堆。下面的 C 例子程序实现了这一算法。

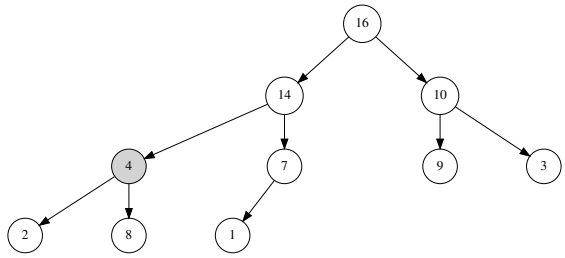
```
typedef int (*Less)(Key, Key);
int less(Key x, Key y) { return x < y; }
int notless(Key x, Key y) { return !less(x, y); }

void heapify(Key* a, int i, int n, Less lt) {
    int l, r, m;
    while (1) {
        l = LEFT(i);
        r = RIGHT(i);
        m = i;
        if (l < n && lt(a[l], a[i]))
            m = l;
```

```
        if (r < n && lt(a[r], a[m]))
            m = r;
        if (m ≠ i) {
            swap(a, i, m);
            i = m;
        } else
            break;
    }
}
```

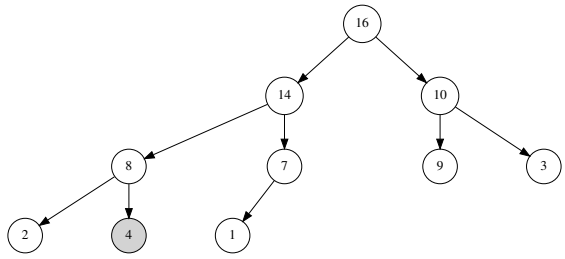
图8.2描述了 HEAPIFY 从索引 2 开始,按照最大堆处理数组 $\{16, 4, 10, 14, 7, 9, 3, 2, 8, 1\}$ 过程中的各个步骤。数组最终变换为 $\{16, 14, 10, 8, 7, 9, 3, 2, 4, 1\}$。



(a) 步骤 1：4、14 和 7 中的最大元素是 14。将 4 和左侧子节点交换；



(b) 步骤 2：2、4 和 8 中的最大元素是 8。将 4 和右侧子节点交换；



(c) 4 为叶子节点。过程结束。

图 8.2: Heapify 的例子，堆为最大堆

8.2.3 构造堆

使用 HEAPIFY 算法，我们可以很方便地从任意数组构造堆。观察完全二叉树各层的节点数：

$1, 2, 4, 8, ..., 2^i, ...$

唯一例外是最后一层，由于树并不一定是满的（完全二叉树不等同于满），最后一层最多含有 2^{p-1} 个节点，其中 $2^p \leq n$，n 是数组的长度。

HEAPIFY 算法对于叶子节点不起任何作用，这是由于所有的叶子节点都已经满足堆性质了。我们可以跳过叶子节点，从第一个分支节点开始执行 HEAPIFY。显然第一个分支节点的索引不大于 $\lfloor n/2 \rfloor$。

根据这一分析，我们可以设计出如下的堆构造算法（以最小堆为例）：

1: **function** BUILD-HEAP(A)
2: $n \leftarrow |A|$
3: **for** $i \leftarrow \lfloor n/2 \rfloor$ down to 1 **do**
4: HEAPIFY(A, i)

虽然 HEAPIFY 算法的复杂度为 $O(\lg n)$,但是 BUILD-HEAP 的复杂度不是 $O(n \lg n)$，而是线性时间 $O(n)$ 的。我们跳过了所有的叶子节点，最多有 1/4 的节点被比较并向下移动一次；最多有 1/8 的节点被比较并向下移动两次；最多有 1/16 的节点被比较并向下移动三次……总共比较和移动次数的上限为：

$$S = n(\frac{1}{4} + 2\frac{1}{8} + 3\frac{1}{16} + ...) \tag{8.1}$$

将两侧都乘以 2：

$$2S = n(\frac{1}{2} + 2\frac{1}{4} + 3\frac{1}{8} + ...) \tag{8.2}$$

用式（8.2）减去式（8.1），我们有：

$$S = n(\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + ...) = n$$

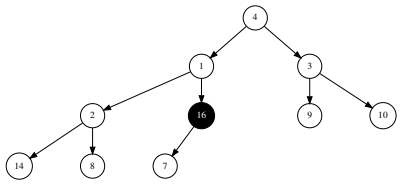
下面的 C 语言例子程序实现了堆构造算法：

```
void build_heap(Key* a, int n, Less lt) {
    int i;
    for (i = (n-1) >> 1; i ≥ 0; --i)
        heapify(a, i, n, lt);
}
```

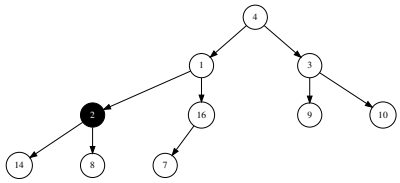
图8.3描述了从数组 $\{4, 1, 3, 2, 16, 9, 10, 14, 8, 7\}$ 构造一个最大堆的各个步骤。黑色节点表示执行 HEAPIFY 时开始的节点；灰色节点表示为了维持堆性质进行交换的节点。

4	1	3	2	16	9	10	14	8	7

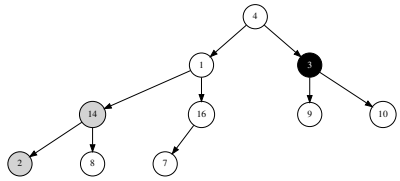
一个无序数组。



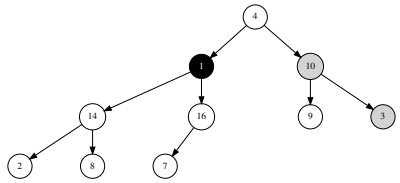
第一步：数组被映射为二叉树。第一个分支节点的值是 16。



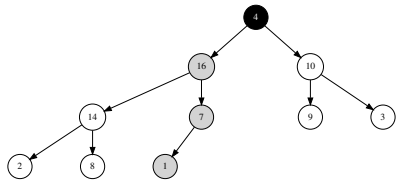
第二步：16 是当前子树中的最大元素，接下来检查元素 2 所在的节点。



第三步：14 是子树中的最大元素，交换 2 和 14；接下来检查元素为 3 的节点。



第四步：10 是子树中的最大元素，交换 10 和 3；接下来检查元素为 1 的节点。



第五步：16 是子树中的最大元素，先交换 16 和 1，接下来交换 1 和 7；此后检查元素为 4 的根节点。

8.2.4 堆的基本操作

堆的通用定义要求我们提供一些基本操作使得用户可以获取或者改变数据。

最重要的操作包括获取顶部元素（查找最小或最大元素），弹出顶部元素，寻找最小（或最大）的前 k 个元素，减小某一元素的值（此操作对应最小堆，最大堆的相应操作是增加某一元素的值），以及插入新元素。

对于用完全二叉树实现的堆，大部分操作的复杂度在最差情况下都是 $O(\lg n)$ 的。有些操作，例如获取顶部元素，仅仅需要常数时间 $O(1)$。

获取顶部元素

在用二叉树实现的堆中，根节点保存了最小（或最大）元素，它对应数组的第一个值。

```
1: function TOP(A)
2:     return A[1]
```

这一简单操作是常数时间 $O(1)$ 的。我们这里省略了对于空堆的错误处理。

弹出堆顶元素

弹出操作比获取顶部元素要复杂一些。我们需要在移除顶部元素后，通过执行 HEAPIFY 算法检查并恢复堆的性质。下面给出了一个简单的实现，但是它的性能较差。

```
1: function POP-SLOW(A)
2:     x ← TOP(A)
3:     REMOVE(A, 1)
4:     if A is not empty then
5:         HEAPIFY(A, 1)
6:     return x
```

这一算法首先用 x 记录下顶部元素，然后将数组中的第一个元素删除，数组的长度减一。如果此后数组不为空，就从新的第一个元素开始执行一次 HEAPIFY。

从长度为 n 的数组中删除第一个元素需要线性时间 $O(n)$。这是因为我们需要将所有剩余的元素依次向前移动一位。这一操作成为了整个算法的瓶颈，使得算法的复杂度升高了。

为了解决这一问题，我们可以交换数组中的第一个和最后一个元素，然后将数组的长度减一。

```
1: function POP(A)
2:     x ← TOP(A)
3:     n ← HEAP-SIZE(A)
4:     EXCHANGE A[1] ↔ A[n]
```

```
    REMOVE(A, n)
    if A is not empty then
        HEAPIFY(A, 1)
    return x
```

从数组的末尾删除最后一个元素仅需要常数时间 $O(1)$，而 HEAPIFY 算法的时间是 $O(\lg n)$ 的。这样整体上弹出操作算法的复杂度为对数时间 $O(\lg n)$。下面的 C 例子程序实现了这一算法[1]。

```
Key pop(Key* a, int n, Less lt) {
    swap(a, 0, --n);
    heapify(a, 0, n, lt);
    return a[n];
}
```

寻找 top k 个元素

使用 pop，可以很方便地找出一组值中的前 k 大个（或前 k 小个）。我们可以构建一个最大堆，然后重复执行 k 次 pop 操作。

```
1: function TOP-K(A, k)
2:     R ← φ
3:     BUILD-HEAP(A)
4:     for i ← 1 to MIN(k, |A|) do
5:         APPEND(R, POP(A))
6:     return R
```

如果 k 超过了数组的长度，我们返回整个数组作为结果。因此上述实现中，我们使用最小值 MIN 函数来决定循环的次数。

下面的 Python 例子程序实现了 top-k 算法：

```
def top_k(x, k, less_p = MIN_HEAP):
    build_heap(x, less_p)
    return [heap_pop(x, less_p) for _ in range(min(k, len(x)))]
```

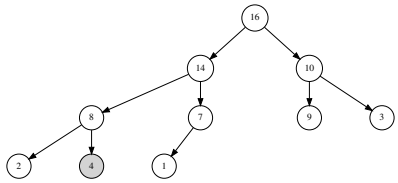
减小 key 值

堆可以用来实现带有优先级的队列，因此需要提供方法来更改堆中的 key 值。例如在实际应用中，为了尽早执行某个任务，我们会提高它的优先级。

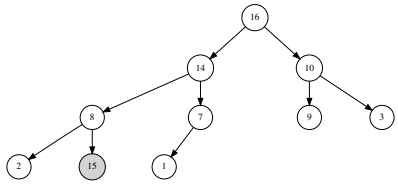
这里我们给出在最小堆中减小 key 的结果，最大堆的相应操作为增加其中的 key。图8.4描述了将最大堆中第 9 个节点从 4 增加到 15 的步骤。

当最小堆中的某个值减小时，可能会违反堆的性质，新的 key 可能比它的祖先小。我们可以定义如下算法来恢复堆的性质。

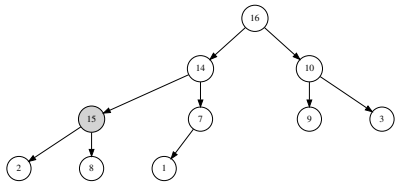
[1]此程序并未删除最后一个元素，而是复用数组的最后一个单元（cell）来存储弹出的结果。



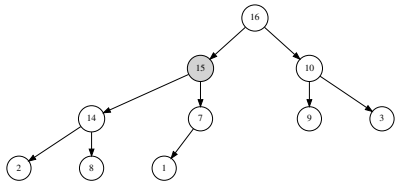
第一步：第 9 个节点的值为 4。



第二步：将 4 增加到 15，大于其父节点的值。



第三步：根据最大堆的性质，交换 8 和 15。



第四步：因为 15 大于父节点的值 14，它们进行交换。此后因为 15 小于 16，处理过程结束。

图 8.4: 增大最大堆中某个值的过程

1: **function** HEAP-FIX(A, i)
2: **while** $i > 1 \wedge A[i] < A[\text{ PARENT}(i)]$ **do**
3: EXCHANGE $A[i] \leftrightarrow A[\text{ PARENT}(i)]$
4: $i \leftarrow \text{PARENT}(i)$

这一算法不断比较当前节点和父节点的值，如果父节点较小，就进行交换。算法自底向上进行检查，直到根节点，或者发现父节点的值较小。

使用这一辅助算法，我们可以实现最小堆中减小 key 的操作。

1: **function** DECREASE-KEY(A, i, k)
2: **if** $k < A[i]$ **then**
3: $A[i] \leftarrow k$
4: HEAP-FIX(A, i)

这一算法仅仅在新 key 比此前的值小时才有效。算法的性能是 $O(\lg n)$ 的。下面的 C 例子程序实现了此算法。

```
void heap_fix(Key* a, int i, Less lt) {
    while (i > 0 && lt(a[i], a[PARENT(i)])) {
        swap(a, i, PARENT(i));
        i = PARENT(i);
    }
}

void decrease_key(Key* a, int i, Key k, Less lt) {
    if (lt(k, a[i])) {
        a[i] = k;
        heap_fix(a, i, lt);
    }
}
```

插入

插入可以用 DECREASE-KEY 来实现 [4]。先构建一个 key 为 ∞ 的新节点。根据最小堆的性质，新节点为数组中的最后一个元素。然后，我们将节点的 key 减小为待插入的值，再使用 DECREASE-KEY 恢复堆性质。

我们也可以直接使用 HEAP-FIX 来实现插入。将待插入的元素直接附加到数组末尾，然后使用 HEAP-FIX 自底向上恢复堆性质。

1: **function** HEAP-PUSH(A, k)
2: APPEND(A, k)
3: HEAP-FIX($A, |A|$)

下面的 Python 例子程序实现了堆插入算法。

```
def heap_insert(x, key, less_p = MIN_HEAP):
    i = len(x)
    x.append(key)
```

```
    heap_fix(x, i, less_p)
```

8.2.5 堆排序

堆排序是堆的一个有趣应用。根据堆的性质，可以很容易地从堆顶获取最小（或最大）元素。我们可以从待排序的元素构建一个堆，然后不断将最小元素弹出直到堆变空。

根据这一想法设计的算法如下：

1: **function** HEAP-SORT(A)
2: $R \leftarrow \phi$
3: BUILD-HEAP(A)
4: **while** $A \neq \phi$ **do**
5: APPEND(R, HEAP-POP(A))
6: **return** R

下面的 Python 例子程序实现了这一定义。

```
def heap_sort(x, less_p = MIN_HEAP):
    res = []
    build_heap(x, less_p)
    while x≠[]:
        res.append(heap_pop(x, less_p))
    return res
```

若待排序的元素有 n 个，通过 BUILD-HEAP 构建堆的复杂度是 $O(n)$ 的。由于 pop 操作的复杂度为 $O(\lg n)$，并且共执行了 n 次。因此堆排序的总体的复杂度为 $O(n \lg n)$。由于我们使用了另外一个列表存放排序结果，因此需要的空间为 $O(n)$。

Robert. W. Floyd 给出了一个堆排序的高效实现。思路是构建一个最大堆而不是最小堆。这样第一个元素就是最大的。接下来，将最大的元素和数组末尾的元素交换，这样最大元素就存储到了排序后的正确位置。而原来在末尾的元素变成了新的堆顶。这会违反堆的性质，我们需要将堆的大小减一，然后执行 HEAPIFY 恢复堆的性质。我们重复这一过程，直到堆中仅剩下一个元素。

1: **function** HEAP-SORT(A)
2: BUILD-MAX-HEAP(A)
3: **while** $|A| > 1$ **do**
4: EXCHANGE $A[1] \leftrightarrow A[n]$
5: $|A| \leftarrow |A| - 1$
6: HEAPIFY($A, 1$)

这一算法是原地排序的，无需使用额外的空间来存储结果。下面的 C 例子程序实现了此算法。

```
void heap_sort(Key* a, int n) {
```

```
    build_heap(a, n, notless);
    while(n > 1) {
        swap(a, 0, --n);
        heapify(a, 0, n, notless);
    }
}
```

练习 8.1

- 考虑另外一种实现原地堆排序的方法：第一步先从待排序数组构建一个最小堆 A，此时，第一个元素 a_1 已经在正确的位置了。接下来，将剩余的元素 $\{a_2, a_3, ..., a_n\}$ 当成一个新的堆，并从 a_2 开始执行 HEAPIFY。重复这一从左向右的步骤完成排序。下面的 C 语言代码实现了这一想法。这一方法正确么？如果正确，请给出证明，如果错误，请指出原因。

```
void heap_sort(Key* a, int n) {
    build_heap(a, n, less);
    while(--n)
        heapify(++a, 0, n, less);
}
```

- 基于同样的道理，我们可以通过自左向右执行 k 遍 HEAPIFY 来实现原地修改的 top-k 算法么？如下面的 C 语言例子代码所示：

```
int tops(int k, Key* a, int n, Less lt) {
    build_heap(a, n, lt);
    for (k = MIN(k, n) - 1; k; --k)
        heapify(++a, 0, --n, lt);
    return k;
}
```

8.3 左偏堆和 skew 堆 显式的二叉堆

人们很自然会问：如果不使用数组，有没有可能使用普通的二叉树来实现堆？

如果使用显式的二叉树作为堆的底层数据结构，我们必须解决一些问题。第一个问题是关于 HEAP-POP 和 DELETE-MIN 操作的。考虑图8.5所示的二叉树 (L, k, R)，其中 L、k、R 分别表示左子树、key 和右子树。

如果 k 是一个最小堆的顶部元素，所有左右子树中的元素都大于 k。k 被弹出后，只剩下左右子树。我们需要把它们合并为一棵新树。由于合并后必须保持堆的性质，新的根节点必须仍保存剩余元素中的最小元素。

因为左右子树也都是符合堆性质的二叉树，我们可以立即给出两个特殊情况下的结果：

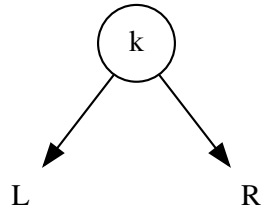


图 8.5: 二叉树，所有子节点中的元素都大于 k

$$merge(H_1, H_2) = \begin{cases} H_2 & : & H_1 = \phi \\ H_1 & : & H_2 = \phi \\ ? & : & otherwise \end{cases}$$

其中 ϕ 表示空堆。如果左右子树都不为空，因为它们都满足堆的性质，因此各自的根节点都保存了最小的元素。我们可以比较两棵树的根，选择较小的一个作为堆合并后的根。

举例来说，令 $L = (A, x, B)$、$R = (A', y, B')$，其中 A、A'、B、B' 都是子树，如果 $x < y$，x 就将是新的根。我们或者可以保留 A，然后递归地将 B 和 R 合并；或者保留 B，然后递归地合并 A 和 R。新的堆可以为下面之一：

- $(merge(A, R), x, B)$
- $(A, x, merge(B, R))$

两个都是正确的结果，为了简单，我们可以总选择右侧的子树进行合并。左偏堆（*Leftist* heap）就是基于这一思想实现的。

8.3.1 定义

使用左偏树实现的堆称为左偏堆。左偏树最早由 C. A. Crane 于 1972 年引入[43]。

Rank（S-值）

左偏树中每个节点都定义了一个 Rank 值（或称 S 值）。Rank 被定义为到达最近的外部节点的距离。其中外部节点指空节点 NIL，例如叶子节点的子节点就是外部节点。

如图8.6所示，NIL 的 Rank 被定义为 0。考虑根节点 4，最近的叶子节点为 8，所以根节点的 Rank 为 2。因为节点 6 和节点 8 都是叶子节点，所以它们的 Rank 为

1。虽然节点 5 则左子树不为空，但是它的右子树是空节点，因此 Rank 值，也就是到达 NIL 的最短距离仍然为 1。

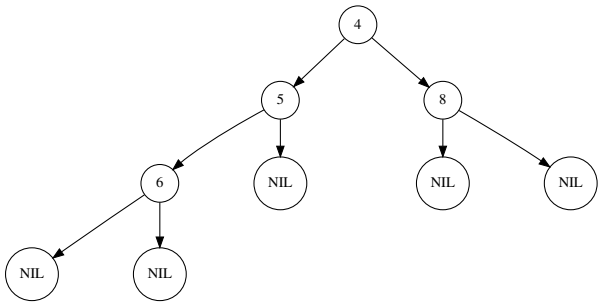


图 8.6: $rank(4) = 2$、$rank(6) = rank(8) = rank(5) = 1$

左偏性质

使用 Rank，我们可以定义合并时的策略：

- 总是合并右侧子树。记新的右侧子树的 Rank 值为 r_r；
- 比较左右子树的 Rank 值，记左子树的 Rank 值为 r_l，如果 $r_l < r_r$，就交换左右子树。

我们称上面的合并策略为“左偏性质”。概括来说，在一棵左偏树中，到某个外部节点的最短距离总是在右侧。

左偏树总是趋向不平衡，但是它可以维护一条重要的性质，如下面的定理所述：

定理 8.3.1. 若一棵左偏树 T 包含 n 个内部节点，从根节点到达最右侧的外部节点的路径上最多含有 $\lfloor \log(n+1) \rfloor$ 个节点。

我们这里省略了此定理的证明，读者可以参考 [44]，[45] 来了解证明的过程。根据此定理，沿着这一路径进行操作的算法，都可以保证 $O(\lg n)$ 的复杂度。

我们可以在二叉树定义的基础上增加一个 Rank 值来定义左偏树。记非空的左偏树为 (r, k, L, R)。下面的 Haskell 例子程序定义了左偏树。

```
data LHeap a = E -- 空
             | Node Int a (LHeap a) (LHeap a) -- Rank、元素、左、右子树
```

我们定义空树的 Rank 为 0，否则，我们通过读取新增加的变量 r 来获得 Rank 值。下面的 $rank(H)$ 函数可以获取任一情况下的值。

$$rank(H) = \begin{cases} 0 & : \quad H = \phi \\ r & : \quad H = (r, k, L, R) \end{cases} \tag{8.3}$$

对应的 Haskell 例子程序如下：

```
rank E = 0
rank (Node r _ _ _) = r
```

方便起见，我们以后将 $rank(H)$ 简记为 r_H。

8.3.2 合并

为了实现合并操作，我们需要显定义一个算法用以比较左右子树的 Rank 值，并适当地进行子树的交换。

$$mk(k, A, B) = \begin{cases} (r_A + 1, k, B, A) & : \quad r_A < r_B \\ (r_B + 1, k, A, B) & : \quad otherwise \end{cases} \tag{8.4}$$

这一函数接受三个参数，一个 key 和两棵子树 A、B。如果 A 的 Rank 较小，算法就用 B 作为左子树，A 作为右子树来构建一棵较大的树。然后它将 A 的 Rank 加一作为这棵新树的 Rank 值，即新树的 Rank 值为 $r_A + 1$；否则，如果 B 的 Rank 较小，就用 A 作为左子树，B 作为右子树。新树的 Rank 值为 $r_B + 1$。

由于构造新树的时候，我们在顶部增加了一个新的 key。所以 Rank 的值会增长 1。

给定两个左偏堆 H_1 和 H_2，记它们的 key 和左右子树分别为：k_1, L_1, R_1 和 k_2, L_2, R_2。下面的 $merge(H_1, H_2)$ 函数定义了合并算法：

$$merge(H_1, H_2) = \begin{cases} H_2 & : \quad H_1 = \phi \\ H_1 & : \quad H_2 = \phi \\ mk(k_1, L_1, merge(R_1, H_2)) & : \quad k_1 < k_2 \\ mk(k_2, L_2, merge(H_1, R_2)) & : \quad otherwise \end{cases} \tag{8.5}$$

函数 $merge$ 总是在右子树上进行递归调用，因此左偏的性质得以保持。这样就保证了算法的复杂度为 $O(\lg n)$。

下面的 Haskell 例子代码实现了合并算法。

```
merge E h = h
merge h E = h
merge h1@(Node _ x l r) h2@(Node _ y l' r') =
    if x < y then makeNode x l (merge r h2)
    else makeNode y l' (merge h1 r')

makeNode x a b = if rank a < rank b then Node (rank a + 1) x b a
                 else Node (rank b + 1) x a b
```

合并由数组表示的二叉堆

使用数组表示的二叉堆在大多数情况下速度都很快。并且很和现代计算机的高速缓存技术（cache）配合良好。但是合并操作的算法复杂度却为线性时间 $O(n)$。通常的实现是将两个数组连接起来，然后在连接后的结果上重新构建堆 [50]。

```
1: function MERGE-HEAP(A, B)
2:     C ← CONCAT(A, B)
3:     BUILD-HEAP(C)
```

8.3.3 基本堆操作

使用此前定义的 $merge$ 算法，我们可以实现很多基本的堆操作。

获取顶部元素和弹出操作

由于最小的元素总是存储于根节点，我们可以在常数时间 $O(1)$ 内获取到堆的顶部元素。下式从非空的堆 $H = (r, k, L, R)$ 中获取顶部元素。我们忽略了树为空时的错误处理。

$$top(H) = k \tag{8.6}$$

为了实现弹出操作，我们首先将顶部元素删除，然后将左右子树合并为一个新的堆。

$$pop(H) = merge(L, R) \tag{8.7}$$

由于弹出算法的实现直接调用了 $merge$ 函数，因此左偏树弹出操作的复杂度也是 $O(\lg n)$。

插入

我们可以从待插入的元素构建一棵只有一个叶子节点的树，然后将它和已有的左偏树合并到一起。

$$insert(H, k) = merge(H, (1, k, \phi, \phi)) \tag{8.8}$$

显然，由于直接调用 $merge$ 函数，这一算法的复杂度也是 $O(\lg n)$。

使用插入操作，我们可以很容易地将一个列表中的元素依次插入到左偏堆中。下面的构造算法使用了 folding。

$$build(L) = fold(insert, \phi, L) \tag{8.9}$$

图8.7给出另一个构造左偏树的例子。

下面的 Haskell 例子程序实现了上面的各个左偏树操作。

```
insert h x = merge (Node 1 x E E) h

findMin (Node _ x _ _) = x

deleteMin (Node _ _ l r) = merge l r

fromList = foldl insert E
```

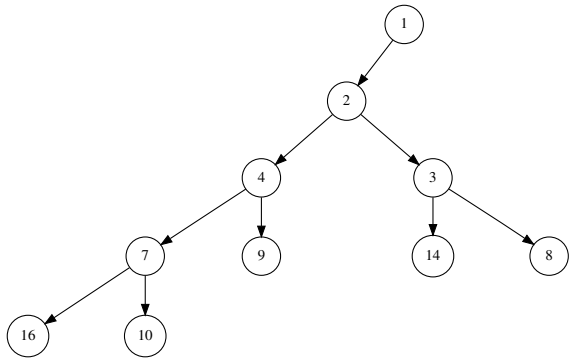


图 8.7: 从列表 $\{9, 4, 16, 7, 10, 2, 14, 3, 8, 1\}$ 构造左偏树

8.3.4 使用左偏堆实现堆排序

使用堆的基本操作，我们可以给出堆排序的实现。给定一个序列，我们首先将它转换成一个左偏堆，然后不断从堆中取得最小元素。

$$sort(L) = heapSort(build(L)) \tag{8.10}$$

$$heapSort(H) = \begin{cases} \phi & : \quad H = \phi \\ \{top(H)\} \cup heapSort(pop(H)) & : \quad otherwise \end{cases} \tag{8.11}$$

因为弹出操作的复杂度是对数时间的，并且被调用了 n 次，因此排序的总体复杂度为 $O(n \lg n)$。下面的 Haskell 例子程序实现了左偏树的堆排序。

```
heapSort = hsort ∘ fromList where
    hsort E = []
    hsort h = (findMin h):(hsort $ deleteMin h)
```

8.3.5 Skew 堆

左偏堆在某些情况下会产生很不平衡的结构。图8.8给出了一个例子，依次将序列 $\{16, 14, 10, 8, 7, 9, 3, 2, 4, 1\}$ 中的元素插入到左偏堆。

Skew 堆（或称自调整堆）既简化了左偏堆的实现，又提高了平衡性 [46]、[47]。

在构造左偏堆的时候，如果左侧的 Rank 值小于右侧的，我们就交换左右子树。但是这一“比较 交换”的策略在 merge 时不能很好处理某一分支为叶子节点的情况。这是因为，不管这棵树有多大，它的 Rank 值总为 1。一种“简单粗暴”的解决方式是，每次合并，我们都交换左右子树。这就是 Skew 堆的原理。

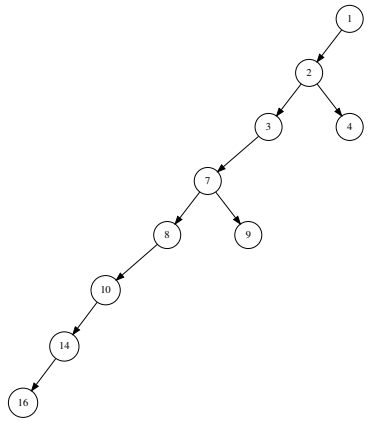


图 8.8: 从序列 $\{16, 14, 10, 8, 7, 9, 3, 2, 4, 1\}$ 构造的左偏堆很不平衡

Skew 堆的定义

Skew 堆是由 skew 树实现的堆。Skew 树是一种特殊的二叉树。最小的元素保存在根节点，每棵子树也都是一棵 skew 树。

Skew 树无需保存 Rank 值（或 S 值）。我们可以直接复用二叉树的定义。树或者为空，或者记为前序形式 (k, L, R)。下面的 Haskell 例子代码定义了 skew 树。

```
data SHeap a = E -- 空
             | Node a (SHeap a) (SHeap a) -- 元素、左、右
```

合并

合并算法被大幅度简化：当合并两棵非空 skew 树时，我们比较根节点，选择较小的作为新的根。然后把含有较大元素的树合并到某一子树上。最后再把左右子树交换。记两棵非空子树为：$H_1 = (k_1, L_1, R_1)$ 和 $H_2 = (k_2, L_2, R_2)$。若 $k_1 < k_2$，选择 k_1 作为新的根。我们既可以将 H_2 和 L_1 合并，也可以将 H_2 和 R_1 合并。不失一般性，我们合并到 R_1 上。然后交换左右子树，最后的结果为 $(k_1, merge(R_1, H_2), L_1)$。考虑边界情况，最终的算法定义如下：

$$merge(H_1, H_2) = \begin{cases} H_1 & : & H_2 = \phi \\ H_2 & : & H_1 = \phi \\ (k_1, merge(R_1, H_2), L_1) & : & k_1 < k_2 \\ (k_2, merge(H_1, R_2), L_2) & : & otherwise \end{cases} \tag{8.12}$$

其他的操作，包括插入，获取顶部元素和弹出都和左偏树一样通过调用 merge 来实现。唯一的不同是我们不再需要 Rank 了。

下面的 Haskell 例子程序实现了 skew 堆。

```
merge E h = h
merge h E = h
merge h1@(Node x l r) h2@(Node y l' r') =
    if x < y then Node x (merge r h2) l
    else Node y (merge h1 r') l'

insert h x = merge (Node x E E) h

findMin (Node x _ _) = x

deleteMin (Node _ l r) = merge l r
```

即使我们用 skew 堆处理已序序列，结果仍然是一棵较平衡的二叉树，如图8.9所示。

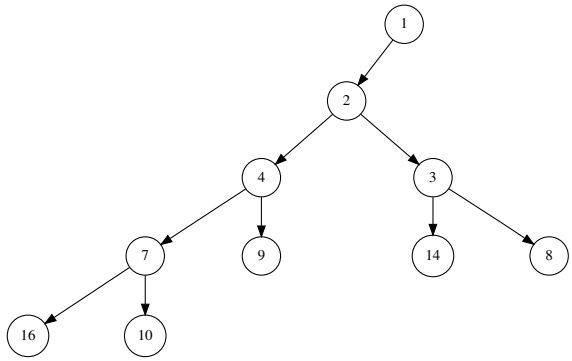


图 8.9: 用已序序列 $\{1, 2, ..., 10\}$ 构造的 skew 树仍然比较平衡

8.4 伸展堆

左偏堆和 skew 堆说明，使用二叉树完全可以实现堆这种数据结构。而且 skew 堆还给出了一种解决树平衡的方法。本节介绍的伸展堆给出了另外一种改善平衡性的方法。

左偏堆和 skew 堆使用的树都不是二叉搜索树（BST）。如果我们将底层的数据结构换成二叉搜索树，最小（或最大）元素就不再保存于根节点。我们需要 $O(\lg n)$ 时间来获取最小（或最大）元素。

如果二叉搜索树不平衡，性能会大幅下降。最坏情况下，大部分操作都退化为 $O(n)$。虽然我们可以用红黑树来实现二叉堆，但这太复杂了。伸展树提供了一种轻量级的实现，它的结果可以动态趋向平衡。

8.4.1 定义

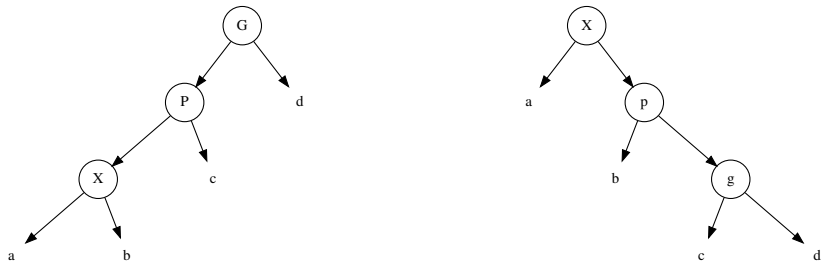
伸展树采用类似于缓存（cache）的策略，它不断将当前正在访问的节点向 top 旋转，这样再次访问的时候就可以更快。我们将这样的操作称为“伸展（splay）”。对于不平衡的二叉搜索树，经过若干次伸展操作后，树会变得越来越平衡。大多数伸展树操作的均摊（amortized）性能都是 $O(\lg n)$ 的。Daniel Dominic Sleator 和 Robert Endre Tarjan 在 1985 年最早引入了伸展树 [48][49]。

伸展操作

有两种方法可以实现伸展操作。第一种需要处理较多的情况，但可以很容易地使用模式匹配（pattern matching）来实现；第二种具备统一的形式，但是实现较为复杂。

记当前正在访问的节点为 X，它的父节点为 P，如果存在祖父节点，则记为 G。伸展操作分为三各步骤，每个步骤有两个对称的情况，为了节省篇幅，我们只给出每步中的一种情况。

- Zig-zig 步骤，如图8.10所示，X 和 P 都是左子树或者 X 和 P 都是右子树。我们通过两次旋转，将 X 变成根节点。



(a) X 和 P 都是左子树或者 X 和 P 都是右子树。

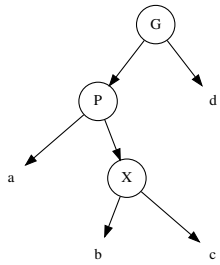
(b) X 经过两次旋转后成为了根节点。

图 8.10: Zig-zig 情况

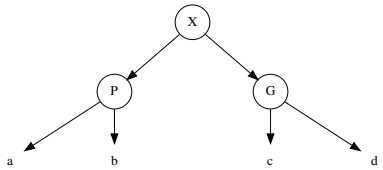
- Zig-zag 步骤，如图8.11所示，X 和 P 一棵是左子树另一棵是右子树。经过旋转，X 变成根节点，P 和 G 变成了兄弟节点。

- Zig 步骤，如图8.12所示，这种情况下，P 是根节点，经过旋转，X 变成了根节点。这是伸展操作的最后一步。

虽然有 6 种不同的情况，但是它们可以很容易地用模式匹配来处理。记非空二叉

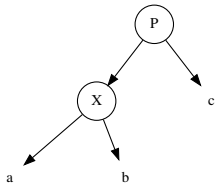


(a) X 和 P 一棵是左子树另一棵是右子树。

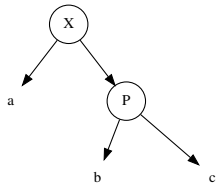


(b) X 成为根节点，P 和 G 变成了兄弟节点。

图 8.11: Zig-zag 情况



(a) P 是根节点。



(b) 通过旋转将 X 变为根节点。

图 8.12: Zig 情况

树为 $T = (L, k, R)$，当访问树中的节点 Y 时，伸展操作可以定义如下：

$$splay(T, X) = \begin{cases} (a, X, (b, P, (c, G, d))) & : & T = (((a, X, b), P, c), G, d), X = Y \\ (((a, G, b), P, c), X, d) & : & T = (a, G, (b, P, (c, X, d))), X = Y \\ ((a, P, b), X, (c, G, d)) & : & T = (a, P, (b, X, c), G, d), X = Y \\ ((a, G, b), X, (c, P, d)) & : & T = (a, G, ((b, X, c), P, d)), X = Y \\ (a, X, (b, P, c)) & : & T = ((a, X, b), P, c), X = Y \\ ((a, P, b), X, c) & : & T = (a, P, (b, X, c)), X = Y \\ T & : & otherwise \end{cases} \tag{8.13}$$

前两条子式处理“zig-zig”情况；接下来的两条子式处理“zig-zag”情况；最后两条子式处理“zig”情况。其他情况下，树都保持不变。

下面的 Haskell 例子程序实现了伸展操作。

```
data STree a = E -- 空
             | Node (STree a) a (STree a) -- left, key, right

-- zig-zig
splay t@(Node (Node (Node a x b) p c) g d) y =
    if x == y then Node a x (Node b p (Node c g d)) else t
splay t@(Node a g (Node b p (Node c x d))) y =
    if x == y then Node (Node (Node a g b) p c) x d else t
-- zig-zag
splay t@(Node (Node a p (Node b x c)) g d) y =
    if x == y then Node (Node a p b) x (Node c g d) else t
splay t@(Node a g (Node (Node b x c) p d)) y =
    if x == y then Node (Node a g b) x (Node c p d) else t
-- zig
splay t@(Node (Node a x b) p c) y = if x == y then Node a x (Node b p c) else t
splay t@(Node a p (Node b x c)) y = if x == y then Node (Node a p b) x c else t
-- 否则
splay t _ = t
```

每次插入新 key 时，我们就执行伸展操作来调整树的平衡性。如果树为空，结果为一个叶子节点；否则我们比较待插入的 key 和根节点，如果待插入的 key 较小，就将其递归插入左子树，然后执行伸展操作；否则将 key 插入右子树，再执行伸展操作。

$$insert(T, x) = \begin{cases} (\phi, x, \phi) & : & T = \phi \\ splay((insert(L, x), k, R), x) & : & T = (L, k, R), x < k \\ splay(L, k, insert(R, x)) & : & otherwise \end{cases} \tag{8.14}$$

下面的 Haskell 程序实现了插入算法。

```
insert E y = Node E y E
insert (Node l x r) y
    | x > y     = splay (Node (insert l y) x r) y
    | otherwise = splay (Node l x (insert r y)) y
```

图8.13描述了向伸展树插入逐一插入有序序列 $\{1, 2, ..., 10\}$ 中元素的结果。如果使用普通二叉树，会退化成一条链表。而伸展树则产生比较平衡的结果。

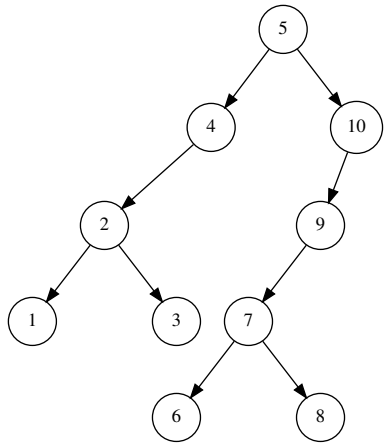


图 8.13: 伸展操作可以改善平衡性

Okasaki 发现了一条简单的伸展操作规则 [3]：每次连续向左或者向右访问两次的时候，就旋转两个节点。

根据这一规则，我们可以这样实现伸展：当访问节点 x 的时候（插入、查找或者删除时），如果连续向左侧或者右侧前进两次，我们就将树分割成两部分：L 和 R，其中 L 含有所有小于 x 的节点，R 含有其余的节点。我们可以构建一棵新树（例如插入时），将 x 作为根，L 和 R 分别作为左右子树。分割是递归的，它会对子树也进

行伸展操作。

$$partition(T,p) = \begin{cases} (\phi,\phi) & : & T = \phi \\ (T,\phi) & : & T = (L,k,R) \wedge R = \phi \\ ((L,k,L'),k',A,B) & : & \begin{array}{l} T = (L,k,(L',k',R')) \\ k < p, k' < p \\ (A,B) = partition(R',p) \end{array} \\ ((L,k,A),(B,k',R')) & : & \begin{array}{l} T = (L,K,(L',k',R')) \\ k < p \leq k' \\ (A,B) = partition(L',p) \end{array} \\ (\phi,T) & : & T = (L,k,R) \wedge L = \phi \\ (A,(L',k',(R',k,R)) & : & \begin{array}{l} T = ((L',k',R'),k,R) \\ p \leq k, p \leq k' \\ (A,B) = partition(L',p) \end{array} \\ ((L',k',A),(B,k,R)) & : & \begin{array}{l} T = ((L',k',R'),k,R) \\ k' \leq p \leq k \\ (A,B) = partition(R',p) \end{array} \end{cases} \tag{8.15}$$

函数 $partition(T,p)$ 接受一棵树 T 和一个基准值（pivot）p 为参数。第一条子式处理边界条件。对空树进行分割的结果为一对空的左右子树。否则，记树为 (L,k,R)，我们需要比较基准值 p 和根节点的值 k。如果 $k < p$，分为两种子情况。一种是 R 为空的简单情况，根据二叉搜索树的性质，所有的元素都小于 p，因此结果为 (T,ϕ)。

否则，$R = (L',k',R')$，我们需要递归地用基准值分割 R'，将 R' 中所有小于 p 的元素放入树 A，其余元素放入树 B。结果为一对树，其中一棵为 $((L,k,L'),k',A)$，另一棵为 B。

如果右子树的 key 不小于基准值，我们递归地用基准值分割 L' 得到结果 (A,B)。最终的结果为一对树，一棵是 (L,k,A)，另一棵是 (B,k',R')。当 $p \leq k$ 时，情况是对称的，由最后的三条子式处理。

下面的 Haskell 例子程序实现了分割算法。

```
partition E _ = (E, E)
partition t@(Node l x r) y
    | x < y =
        case r of
          E → (t, E)
          Node l' x' r' →
              if x' < y then
```

```
                let (small, big) = partition r' y in
                (Node (Node l x l') x' small, big)
            else
                let (small, big) = partition l' y in
                (Node l x small, Node big x' r')
    | otherwise =
        case l of
          E → (E, t)
          Node l' x' r' →
            if y < x' then
                let (small, big) = partition l' y in
                (small, Node l' x' (Node r' x r))
            else
                let (small, big) = partition r' y in
                (Node l' x' small, Node big x r)
```

我们可以用 $partition$ 实现插入算法。当向一个伸展堆 T 插入一个新元素 k 时，我们先将堆分割为两棵子树 L 和 R。其中 L 含有所有小于 k 的节点，而 R 含有剩余的部分。然后我们构建一棵新树，使用 k 作为根，L 和 R 作为子树。

$$insert(T, k) = (L, k, R), (L, R) = partition(T, k) \tag{8.16}$$

对应的 Haskell 例子程序如下：

```
insert t x = Node small x big where (small, big) = partition t x
```

获取和弹出顶部元素

由于伸展树本质上是二叉搜索树，最小的元素存储于最左侧的节点中。我们需要不断向左遍历以获取顶部元素。记非空的树为 $T = (L, k, R)$，$top(T)$ 函数可以定义如下：

$$top(T) = \begin{cases} k & : & L = \phi \\ top(L) & : & otherwise \end{cases} \tag{8.17}$$

这实际上就是二叉搜索树的 $min(T)$ 算法。

对于弹出操作，算法需要将最小元素删除。每当连续向左访问两次，就执行一次伸展操作。

$$pop(T) = \begin{cases} R & : & T = (\phi, k, R) \\ (R', k, R) & : & T = ((\phi, k', R'), k, R) \\ (pop(L'), k', (R', k, R)) & : & T = ((L', k', R'), k, R) \end{cases} \tag{8.18}$$

注意这里的第三条子式实际上执行了伸展操作，它并没有显式地调用 $partition$ 函数，而是直接使用了二叉搜索树的性质。

因为伸展树是平衡的，top 和 pop 操作的性能都是 $O(\lg n)$。

下面的 Haskell 例子程序实现了 top 和 pop 操作。

```
findMin (Node E x _) = x
findMin (Node l x _) = findMin l

deleteMin (Node E x r) = r
deleteMin (Node (Node E x' r') x r) = Node r' x r
deleteMin (Node (Node l' x' r') x r) = Node (deleteMin l') x' (Node r' x r)
```

合并

合并是堆的一个重要操作，它被广泛用于图算法。通过使用 $partition$ 函数，我们可以实现一个 $O(\lg n)$ 时间的合并算法。

当合并两棵伸展树时，如果它们都不为空，我们可以将第一棵树的根节点作为新的根，然后将其作为基准值分割第二棵树。此后，我们递归地将第一棵树的子树合并。算法定义如下：

$$merge(T_1, T_2) = \begin{cases} T_2 & : & T_1 = \phi \\ (merge(L, A), k, merge(R, B)) & : & T_1 = (L, k, R), (A, B) = partition(T_2, k) \end{cases} \tag{8.19}$$

如果第一个堆为空，结果显然为第二个堆。否则，记第一个堆为 (L, k, R)，我们使用 k 作为基准值分割 T_2 得到结果 (A, B)，其中 A 包含 T_2 中所有小于 k 的节点，而 B 包含其余节点。我们接下来递归地将 A 和 L 合并为新的左子树，将 B 和 R 合并为右子树。

这一定义可以翻译为下面的 Haskell 例子程序。

```
merge E t = t
merge (Node l x r) t = Node (merge l l') x (merge r r')
    where (l', r') = partition t x
```

8.4.2 堆排序

由于伸展堆的内部实现对于通用堆的接口完全透明，我们可以完全复用此前的堆排序定义。也就是说堆排序的算法也是通用的，它不依赖于底层的数据结构。

8.5 小结

本章中，我们介绍了通用的二叉堆概念。只要保证堆的性质，我们可以使用任何形式的二叉树来实现堆。

这样的定义并不仅限于使用基于数组的二叉堆，它也包含使用其他二叉树形式的堆如左偏堆、skew 堆和伸展堆。基于数组的二叉堆易于用命令式的方式实现。它将一棵完全二叉树映射为数组的随机访问，我们很难找到和它直接对应的纯函数式实现。

但是，我们可以通过使用显式的二叉树来实现纯函数式的二叉堆。大部分的操作在最坏情况下也可以达到 $O(\lg n)$ 的性能。有些操作的分摊性能甚至可以达到 $O(1)$。Okasaki 在 [3] 中给出了这些数据结构的详细分析。

我们仅在本章中给出了左偏堆、skew 堆和伸展堆的纯函数式实现。它们也都支持命令式实现。

人们很自然希望能将二叉树扩展到 k 叉树，这样就会得到其他重要的数据结构如二项式（Binomial）堆、斐波那契（Fibonacci）堆和配对（pairing）堆。我们将在后面的章节加以介绍。

练习 8.2

- 用命令式的方式实现左偏堆、skew 堆和伸展堆。

第九章　从吃葡萄到世界杯，选择排序的进化

9.1　简介

我们此前介绍了排序中的“hello world” 插入排序算法。本章中，我们介绍另外一种直观的排序方法　选择排序。最基本的选择排序在性能上不如分而治之的排序算法，如快速排序和归并排序。我们将会分析为什么选择排序的速度慢，并且从不同的角度改进它，最终进化到堆排序，从而达到基于比较的排序算法的性能上限 $O(n \lg n)$。

选择排序的思想在日常生活中很常见。考虑一个小孩要吃掉一串葡萄。我们通常会发现两种类型的小孩，一种属于“乐观型”，每次吃掉最大的一颗；另一种属于“悲观型”，每次总吃掉最小的一颗。

第一种小孩实际上按照由大到小的顺序吃葡萄；第二种按照由小到大的顺序吃葡萄。实际上，孩子们把葡萄按照大小进行了排序，并且使用了选择排序的思想。

图 9.1: 总挑出最小的葡萄

选择排序的算法可以描述如下。

为了将元素排序：

- 简单情况：如果序列为空，排序结果也为空；
- 否则，我们找到最小的元素，将其附加到结果的后面。

注意上面描述的算法将元素按照升序排序；如果每次选择最大的元素，则排序结果是降序的。我们稍后会介绍如何将比较方法作为参数传入。

这一描述可以形式化为下面的公式。

$$sort(A) = \begin{cases} \phi & : \quad A = \phi \\ \{m\} \cup sort(A') & : \quad otherwise \end{cases} \tag{9.1}$$

其中 m 是序列 A 中的最小元素，A' 是除 m 外的剩余元素。

$$m = min(A)$$
$$A' = A - \{m\}$$

我们并不限定序列的具体数据结构。通常在命令式的环境中 A 是数组，在函数式环境中，A 是单向链表。我们在后面将会看到，A 甚至可以是其他数据结构。

这一算法也可以用命令式的方式给出：

```
1: function SORT(A)
2:     X ← φ
3:     while A ≠ φ do
4:         x ← MIN(A)
5:         A ← DEL(A, x)
6:         X ← APPEND(X, x)
7:     return X
```

图9.2描述了选择排序的过程。

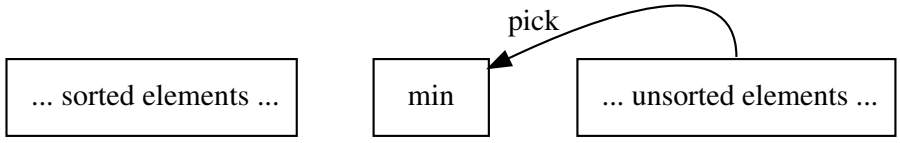


图 9.2: 左侧部分为已排序的元素，算法不断从剩余部分选择最小元素附加的左侧的尾部

到目前为止，我们将“吃葡萄”的过程转换成了算法，但是没有考虑任何时间和空间上的开销。我们将结果保存在一个列表 X 中，把从剩余部分选择出的元素取出并添加到 X 的末尾。实际上，我们可以复用 A 中的空间实现原地排序。

具体方法是我们将最小的元素保存在 A 的第一个单元（cell）中（若 A 为数组，我们用单元来表示存储单位，若 A 为链表，我们用节点来表示存储单位），将第二小的元素保存在下一个单元中，接下来是第三个单元……

可以用交换的方法来实现这一排序策略。当我们找到第 i 小的元素后，我们将它和第 i 个位置的单元交换。

```
1: function SORT(A)
2:     for i ← 1 to |A| do
```

3: $m \leftarrow \text{MIN}(A[i...])$
4: EXCHANGE $A[i] \leftrightarrow m$

记 $A = \{a_1, a_2, ..., a_n\}$，任何时候，当我们处理第 i 个元素时，所有在 i 之前的部分 $\{a_1, a_2, ..., a_{i-1}\}$ 都已经排好序了。我们找到 $\{a_i, a_{i+1}, ..., a_n\}$ 中的最小元素，然后将其和 a_i 交换，这样第 i 个位置就保存了正确的元素。重复这一过程直到最后一个元素。

图9.3描述了这一思路。

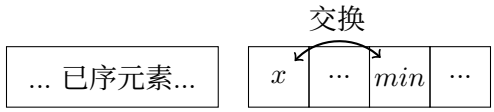


图 9.3: 左侧部分为已序元素，不断从剩余元素中找到最小的存储到正确的位置

9.2 查找最小元素

我们尚未完全实现选择排序，如何查找最小（或最大）的元素仍然是一个黑盒子。小孩子们是如何找到最小或最大的一粒葡萄的？这对于计算机算法来说也是一个有趣的题目。

虽然不够快，但是最简单的方法是对所有的元素进行一次扫描。存在几种不同的方法来实现扫描的过程。假设要选出最大的一粒葡萄。我们任选一颗，将它和另外一颗比较，然后留下较大的；此后我们再选另外一颗，和手里留下的这颗比较，并留下最大的。重复这一“选取　比较”的过程直到比较完所有的葡萄。

实践一下，我们会发现，如果不对比较过的葡萄做记号，很快就会搞乱，我们会忘记哪些葡萄比较过了，哪些还没有。有两种方法可以解决这一问题，它们分别适合不同的数据结构。

9.2.1 标记

第一种方法是将葡萄标记上编号，例如：$\{1, 2, ..., n\}$，然后按照编号的顺序进行比较。先比较第一号葡萄和第二号葡萄，选择较大的留下；然后用第三号葡萄做比较……重复这一步骤直到第 n 号葡萄。标记方法很适合处理数组。

```
1: function MIN(A)
2:     m ← A[1]
3:     for i ← 2 to |A| do
4:         if A[i] < m then
5:             m ← A[i]
6:     return m
```

使用 MIN 函数，我们可以最终完成基本的选择排序（没有任何空间和时间上的优化）。

上述查找最小元素的方法返回的是元素的值而非它的位置（葡萄的编号），如果要实现原地排序，还需要做一些调整。某些语言，如 C++ 支持返回引用作为结果，因此可以直接实现元素的交换。

```
template<typename T>
T& min(T* from, T* to) {
    T* m;
    for (m = from++; from ≠ to; ++from)
        if (*from < *m)
            m = from;
    return *m;
}

template<typename T>
void ssort(T* xs, int n) {
    for (int i = 0; i < n; ++i)
        std::swap(xs[i], min(xs+i, xs+n));
}
```

在不支持引用语意的环境中，可以返回元素的位置，而不是值作为结果。

```
1: function MIN-AT(A)
2:     m ← FIRST-INDEX(A)
3:     for i ← m + 1 to |A| do
4:         if A[i] < A[m] then
5:             m ← i
6:     return m
```

传入 MIN-AT 的数组实际是 A 的片断 $A[i...]$，我们假设第一个元素 $A[i]$ 是最小的一个，并且逐一检查元素 $A[i+1], A[i+2], ...$。函数 FIRST-INDEX() 用于从参数中获取索引 i。

下面的 Python 例子程序，根据这一思路实现了基本的原地选择排序算法。它直接将数组片断的信息传入，并使用最小元素的位置。

```
def ssort(xs):
    n = len(xs)
    for i in range(n):
        m = min_at(xs, i, n)
        (xs[i], xs[m]) = (xs[m], xs[i])
    return xs

def min_at(xs, i, n):
    m = i;
    for j in range(i+1, n):
        if xs[j] < xs[m]:
            m = j
    return m
```

9.2.2 分组

另外一种方法是将全部葡萄分成两部分：一组包含已经检查并比较过的所有葡萄，另外一组包含剩余未比较过的。记这两组葡萄为 A 和 B、全部的元素（葡萄）为 L。在开始的时候，我们尚未处理任何葡萄，因此 A 为空（ϕ），B 包含全部葡萄。我们可以从 B 中任选两颗葡萄进行比较，然后将较小的一颗放入 A。此后我们不断从 B 中选择葡萄，和上次比较的获胜者（较大的一颗）对比直到 B 变成空。这时，最后的获胜者就是最小的元素，而 A 中包含 $L - \{min(L)\}$，可以用于下一轮的最小值查找。

这一方法的不变关系（invariant）为：在任何时候我们有：$L = A \cup \{m\} \cup B$，其中 m 为目前找到的获胜者。

这一方法无需对葡萄进行编号。它适用于任何可被遍历的数据结构，包括链表等。令 b_1 为非空序列 B 中的任一元素，B' 为除 b_1 外的剩余元素，上述方法可以形式化为如下函数。

$$min'(A, m, B) = \begin{cases} (m, A) & : & B = \phi \\ min'(A \cup \{m\}, b_1, B') & : & b_1 < m \\ min'(A \cup \{b_1\}, m, B') & : & otherwise \end{cases} \tag{9.2}$$

为了选出最小元素，我们调用这一函数并传入空序列 A。选取任何元素（例如第一个）来初始化 m：

$$extractMin(L) = min'(\phi, l_1, L') \tag{9.3}$$

其中 L' 包含 L 中除 l_1 以外的剩余元素。算法 $extractMin$ 不仅查找最小元素，还返回剩余元素以用于接下来的排序。下面的 Haskell 例子程序实现了这一算法。

```
sort [] = []
sort xs = x : sort xs' where
  (x, xs') = extractMin xs

extractMin (x:xs) = min' [] x xs where
  min' ys m [] = (m, ys)
  min' ys m (x:xs) = if m < x then min' (x:ys) m xs else min' (m:ys) x xs
```

第一行处理边界情况，空序列的排序结果仍为空；第二行确保序列中至少含有一个元素，因此 `extractMin` 函数无需再做额外的模式匹配。

有读者认为函数 `min'` 第二个子式应该实现如下：

```
min' ys m (x:xs) = if m < x then min' ys ++ [x] m xs
                           else min' ys ++ [m] x xs
```

否则函数会返回逆序的列表。但这里我们需要用“cons”[1]而不是追加。因为追加操作的复杂度是线性的，和序列 A 的长度成正比，而“cons”是常数时间 $O(1)$ 的。我们并不需要保持待排序元素的顺序，因为排序过程本身就是对顺序的一种改变。

[1]cons 来自 Lisp 语言，表示将一个元素附加到一个链表的头部。详见附录 A。

当然，既保持元素的相对顺序[2]，又能高效地查找最小元素，而不退化成平方复杂度是可以实现的。下面的定义满足了这一限制：

$$extractMin(L) = \begin{cases} (l_1, \phi) & : & |L| = 1 \\ (l_1, L') & : & l_1 < m, (m, L'') = extractMin(L') \\ (m, l_1 \cup L'') & : & otherwise \end{cases} \tag{9.4}$$

如果 L 只含有一个元素（称为 singleton），最小值就是这唯一的元素。否则记 l_1 为 L 中的第一个元素，除 l_1 以外的剩余元素为 L'，即 $L' = \{l_2, l_3, ...\}$。算法递归地从 L' 中查找最小元素，结果记为 (m, L'')，其中 m 是 L' 中的最小元素，而 L'' 包含除 m 外的剩余元素。比较 l_1 和 m 以决定哪个是最终的最小值。

下面的 Haskell 程序实现了这一版本的选择排序。

```
sort [] = []
sort xs = x : sort xs' where
  (x, xs') = extractMin xs

extractMin [x] = (x, [])
extractMin (x:xs) = if x < m then (x, xs) else (m, x:xs') where
  (m, xs') = extractMin xs
```

这里仅仅使用了“cons”操作，而无需“追加”操作。因为算法从右向左处理列表。但是这会有一些额外开销，程序通常需要使用堆栈来保存递归的环境。元素间的相对顺序通过递归来加以保证。读者可以参考附录中关于“尾递归”的内容。

9.2.3 选择排序的性能

无论是标记法，还是分组法都需要在每轮中检查所有未排好的元素以挑选出最小值；总共进行了 n 次挑选。因此处理时间为：$n + (n-1) + (n-2) + ... + 1$ 次比较，即 $\frac{n(n+1)}{2}$。选择排序是平方时间 $O(n^2)$ 的算法。

和此前介绍的插入排序相比，选择排序在最好、最差和平均情况下的性能是相同的，而插入排序在最好情况下性能为线性时间 $O(n)$（元素存储在一个链表中，并且顺序为逆序），最差情况下性能为平方时间 $O(n^2)$。

在接下来的部分，我们将分析为何选择排序的性能较差，并尝试逐步改进它。

练习 9.1

- 选择一门编程语言实现基本的命令式选择排序（非原地排序版本）。和原地排序的算法进行对比，分析时间和空间上的效率。

[2]称为稳定排序 (stable sort)

9.3 细微改进

本节介绍针对选择排序的一些细微改进，首先我们通过参数化，将排序算法变得通用；然后对算法结构进行一些细微调整，使得它更为紧凑。最后我们介绍“鸡尾酒排序”将循环次数减半。

9.3.1 比较方法参数化

在改进性能前，我们先将前面给出的选择排序算法变得更加通用，以便能处理各种排序条件。

我们可以看到存在两种完全相反的排序需要：升序排序和降序排序。对于前者，需要不断查找最小元素；而对于后者，需要不断寻找最大元素。实际应用中，远非这两种情况，还存在各种各样的排序标准，例如按照尺寸、重量、年龄……进行排序。

我们可以将具体的排序标准作为一个比较函数传入选择排序算法，如下：

$$sort(c, L) = \begin{cases} \phi & : \quad L = \phi \\ m \cup sort(c, L'') & : \quad L \neq \phi, (m, L'') = extract(c, L') \end{cases} \tag{9.5}$$

其中 $extract(c, L)$ 的定义为：

$$extract(c, L) = \begin{cases} (l_1, \phi) & : \quad |L| = 1 \\ (l_1, L') & : \quad c(l_1, m), (m, L'') = extract(c, L') \\ (m, \{l_1\} \cup L'') & : \quad \neg c(l_1, m) \end{cases} \tag{9.6}$$

这里 c 是一个比较函数，它接受两个元素，将它们相比并决定哪个的顺序在前面。将“小于”操作 $(<)$ 传入，就是我们此前给出的选择排序实现。

有些环境需要传入全序（total ordering）的比较函数，也就是说比较结果为“小于”、“等于”或者“大于”中的一个。对于选择排序，并不需要这样强的限制，c 只要检查“小于”是否满足即可。但是作为最低要求，比较函数必须满足严格弱序（strict weak ordering）[52]：

- 非自反性（Irreflexivity），对于任何 x，$x < x$ 不成立；
- 非对称性（Asymmetric），对任何 x 和 y，若 $x < y$ 成立，则 $y < x$ 不成立；
- 传递性（Transitivity），对任何 x、y 和 z，若 $x < y$ 且 $y < z$，则 $x < z$ 成立。

下面的 Scheme/Lisp 例子程序实现了参数化的选择排序。Scheme/Lisp 的词法作用域（lexical scope）可以简化比较函数的传递。

```
(define (sel-sort-by ltp? lst)
  (define (ssort lst)
    (if (null? lst)
        lst
        (let ((p (extract-min lst)))
```

```
        (cons (car p) (ssort (cdr p))))))
(define (extract-min lst)
  (if (null? (cdr lst))
      lst
      (let ((p (extract-min (cdr lst))))
        (if (ltp? (car lst) (car p))
            lst
            (cons (car p) (cons (car lst) (cdr p)))))))
(ssort lst))
```

其中 ssort 和 extract-min 都是内部函数，它们都可以直接使用比较函数 ltp?。将 $<$ 传入就可以得到普通的升序排序结果。

```
(sel-sort-by < '(3 1 2 4 5 10 9))
;Value 16: (1 2 3 4 5 9 10)
```

在命令式实现中也可以将比较函数参数化，我们将其作为练习留给读者。

简单起见，本章的后继部分我们仅仅考虑元素的升序排序，除非必要，我们不将比较函数作为参数传入。

9.3.2 细微调整

基本的命令式原地选择排序算法遍历了所有的元素，每次查找出最小的。我们可以把最小值的查找直接实现为一个内重循环，从而使程序变得更加紧凑。

```
1: procedure SORT(A)
2:     for i ← 1 to |A| do
3:         m ← i
4:         for j ← i + 1 to |A| do
5:             if A[i] < A[m] then
6:                 m ← i
7:         EXCHANGE A[i] ↔ A[m]
```

进一步观察，我们需要将 n 个元素排序，当前 $n-1$ 个元素排好后，最后剩下的一个元素，必然是第 n 大的。因此无需再进行一次最小值查找。这样外重循环的次数可以减少一次变成 $n-1$。

还有一处可以进行细微调整，如果第 i 大的元素恰好是 $A[i]$，我们无需进行交换操作。最终实现可以调整为：

```
1: procedure SORT(A)
2:     for i ← 1 to |A| − 1 do
3:         m ← i
4:         for j ← i + 1 to |A| do
5:             if A[i] < A[m] then
6:                 m ← i
```

```
7:        if m ≠ i then
8:            Exchange A[i] ↔ A[m]
```

显然，上面这些调整都对整体的复杂度没有影响。

9.3.3 鸡尾酒排序（Cock-tail sort）

Knuth 给出过一个另一种选择排序的实现 [51]。每次不是查找最小元素，而是最大元素，将其放在末尾位置。如下：

```
1: procedure Sort'(A)
2:     for i ← |A| down-to 2 do
3:         m ← i
4:         for j ← 1 to i − 1 do
5:             if A[m] < A[i] then
6:                 m ← i
7:         Exchange A[i] ↔ A[m]
```

如图13.1所示，任何时候，最右侧的元素都是已序的。算法扫描未排序元素，定位到其中的最大值，然后交换到未排序部分的末尾。

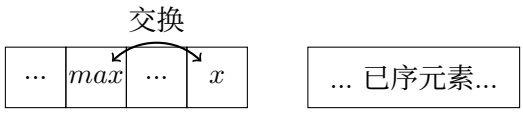


图 9.4: 每次选择最大的元素放到末尾

这一版本的实现说明，每次选择最大的元素也可以实现升序排序。进一步说，我们每次扫描可以同时查找最小值和最大值，分别将最小值放到开头，将最大值放到末尾。这样可以将排序性能略微提高（外重循环次数减半）。我们称这一算法为“鸡尾酒排序”。

```
1: procedure Sort(A)
2:     for i ← 1 to ⌊|A|/2⌋ do
3:         min ← i
4:         max ← |A| + 1 − i
5:         if A[max] < A[min] then
6:             Exchange A[min] ↔ A[max]
7:         for j ← i + 1 to |A| − i do
8:             if A[j] < A[min] then
9:                 min ← j
10:            if A[max] < A[j] then
11:                max ← j
```

12: EXCHANGE $A[i] \leftrightarrow A[min]$
13: EXCHANGE $A[|A| + 1 - i] \leftrightarrow A[max]$

图9.5描述了这一算法，任何时候，左侧部分和右侧部分都包含了已序元素。较小的在左侧，较大的在右侧。算法扫描未排序的部分，定位到最小和最大的两个元素，然后分别将它们交换到未排序部分的开头和末尾。

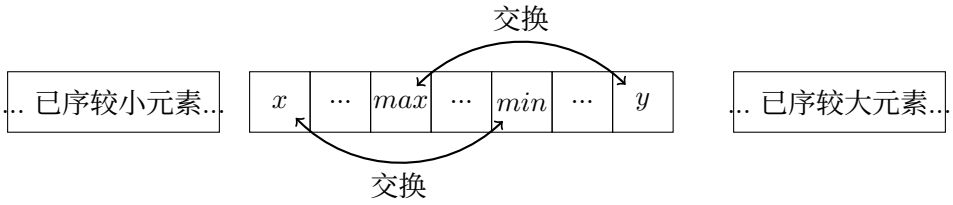


图 9.5: 一次扫描同时定位出最小和最大元素，然后将它们放到正确的位置

注意，在内重循环开始前，如果最右侧的元素小于最左侧的元素，需要将它们交换。这是因为我们的扫描范围不包括两端上的元素。另外一种方法是，在内重循环扫描前，我们将最小和最大元素同时初始化为第一个未排序元素。但是这会产生一个新问题：由于我们在内重循环扫描后要做两次交换操作，有可能第一次交换会改变我们刚刚找到的最大元素或最小元素的位置，这样第二次交换就会产生不正确的结果。我们把如何解决这一问题留给读者作为练习。

下面的 Python 例子程序实现了鸡尾酒排序。

```
def cocktail_sort(xs):
    n = len(xs)
    for i in range(n / 2):
        (mi, ma) = (i, n - 1 -i)
        if xs[ma] < xs[mi]:
            (xs[mi], xs[ma]) = (xs[ma], xs[mi])
        for j in range(i+1, n - 1 - i):
            if xs[j] < xs[mi]:
                mi = j
            if xs[ma] < xs[j]:
                ma = j
        (xs[i], xs[mi]) = (xs[mi], xs[i])
        (xs[n - 1 - i], xs[ma]) = (xs[ma], xs[n - 1 - i])
    return xs
```

鸡尾酒排序也可以用函数式的方式加以实现。一个直观的描述为：

- 边界情况：若待排序的序列为空或者仅含有一个元素，则排序结果为原序列；
- 否则，找到最小和最大值，分别放到开头和结尾位置，然后递归地将剩余元素排序。

算法可以形式化为如下函数。

$$sort(L) = \begin{cases} L & : & |L| \leq 1 \\ \{l_{min}\} \cup sort(L'') \cup \{l_{max}\} & : & otherwise \end{cases} \tag{9.7}$$

其中，函数 $select(L)$ 从序列 L 中抽取出最小值和最大值。

$$(l_{min}, L'', l_{max}) = select(L)$$

注意，最小值实际被直接链结到递归排序结果的前面。语意上是一个常数时间 $O(1)$ 的“cons”操作（参考本书附录 A）。但是最大值被追加到末尾，这一操作的代价较大，通常需要线性时间 $O(n)$。我们稍后再优化它。

函数 $select(L)$ 扫描传入的序列查找最小值和最大值，它的定义为：

$$select(L) = \begin{cases} (min(l_1, l_2), max(l_1, l_2)) & : & L = \{l_1, l_2\} \\ (l_1, \{l_{min}\} \cup L'', l_{max}) & : & l_1 < l_{min} \\ (l_{min}, \{l_{max}\} \cup L'', l_1) & : & l_{max} < l_1 \\ (l_{min}, \{l_1\} \cup L'', l_{max}) & : & otherwise \end{cases} \tag{9.8}$$

其中 $(l_{min}, L'', l_{max}) = select(L')$，$L'$ 是除去 l_1 外的剩余元素。如果序列中仅有两个元素，我们选择较小的最为最小值，较大的作为最大值。剩余序列为空。这是边界情况。否则，我们取出第一个元素 l_1，然后递归地在剩余元素中抽取最小和最大值并和 l_1 做比较以决定最终的结果。

注意所有情况下，我们都不需要做结果列表的追加操作。但是抽取过程需要扫描全部元素，因此性能是线性时间 $O(n)$ 的。

下面的 Haskell 例子程序实现了这一算法。

```
csort [] = []
csort [x] = [x]
csort xs = mi : csort xs' ++ [ma] where
  (mi, xs', ma) = extractMinMax xs

extractMinMax [x, y] = (min x y, [], max x y)
extractMinMax (x:xs) | x < mi = (x, mi:xs', ma)
                     | ma < x = (mi, ma:xs', x)
                     | otherwise = (mi, x:xs', ma)
  where (mi, xs', ma) = extractMinMax xs
```

此前我们指出追加操作的性能开销较大。这一问题可以分两步解决。第一步是将鸡尾酒排序转换为尾递归。左侧包含较小的已序元素，记这一部分为 A；右侧包含较大的已序元素，记这一部分为 B。如图9.5所示。我们用 A 和 B 作为累积器（accumulator），鸡尾酒排序的尾递归实现如下：

$$sort'(A, L, B) = \begin{cases} A \cup L \cup B & : & L = \phi \vee |L| = 1 \\ sort'(A \cup \{l_{min}\}, L'', \{l_{max}\} \cup B) & : & otherwise \end{cases} \tag{9.9}$$

其中 l_{min}、l_{max} 和 L'' 的定义同上。排序开始时，我们传入空的 A 和 B：

$$sort(L) = sort'(\phi, L, \phi)$$

先跳过边界情况，注意到追加操作仅仅发生在 $A \cup \{l_{min}\}$；而 l_{max} 则直接被链结到 B 的前面。每次递归调用都会产生一次追加操作。为了消除它，我们可以将 A 保存为逆序 $\overleftarrow{A}$，这样就可以将 l_{min} 链结到前面而不是执行一次耗时的追加。记 $cons(x, L) = \{x\} \cup L$，$append(L, x) = L \cup \{x\}$，我们有如下等式：

$$\begin{aligned} append(L, x) &= reverse(cons(x, reverse(L))) \\ &= reverse(cons(x, \overleftarrow{L})) \end{aligned} \tag{9.10}$$

最后，我们执行一次反转操作将 $\overleftarrow{A}$ 转换回 A。根据这一思路，算法可进一步被改进如下：

$$sort'(A, L, B) = \begin{cases} reverse(A) \cup B & : \quad L = \phi \\ reverse(\{l_1\} \cup A) \cup B & : \quad |L| = 1 \\ sort'(\{l_{min}\} \cup A, L'', \{l_{max}\} \cup B) & : \end{cases} \tag{9.11}$$

下面的 Haskell 例子程序实现了这一改进。

```
csort' xs = cocktail [] xs [] where
  cocktail as [] bs = reverse as ++ bs
  cocktail as [x] bs = reverse (x:as) ++ bs
  cocktail as xs bs = let (mi, xs', ma) = extractMinMax xs
                      in cocktail (mi:as) xs' (ma:bs)
```

练习 9.2

- 用动态语言和静态语言分别实现基本的选择排序算法，将比较函数用参数传入。在静态语言中，如何声明比较函数的类型才能做到更加通用 (generic)？
- 使用一门编程语言实现 Knuth 给出的选择排序。
- 另一种实现鸡尾酒排序的方法是在内重循环前，假定第 i 个元素既是最小值，也是最大值。内重循环结束后，最小值和最大值被定位出来，需要将最小值交换到第 i 个位置，将最大值交换到第 $|A| + 1 - i$ 个位置。请用一门命令式语言实现这一解法。注意以下特殊的边界情况：
  - $A = \{max, min, ...\}$；
  - $A = \{..., max, min\}$；
  - $A = \{max, ..., min\}$.
- 使用 fold 来实现 $select(L)$ 函数。

9.4 本质改进

虽然鸡尾酒排序将循环次数减半，但是性能仍然是平方级的。如果序列很大，和其他分而治之的排序相比，选择排序的性能明显落后。

为了从本质上改进选择排序，我们必须分析瓶颈出现在哪里。为了通过比较大小进行排序，必须检查所有元素间的大小顺序。因此外重循环是必须的。但是为了选出最小元素，我们必须每次都扫描全部元素么？当查找第一个最小元素时，我们实际遍历了全部的序列，因此我们知道哪些元素相对较小，哪些元素相对较大。

问题在于当查找后继的最小元素时，我们没有复用这些已经获取到的关于相对大小的信息。而是从零开始再次进行遍历。

提高选择排序的关键点在于重用已有的结果。存在几种不同的方法，其中一种是从足球比赛中得到启发的。

9.4.1 锦标赛淘汰法

足球世界杯每四年举办一次。来自各个大洲的 32 支球队最终进入决赛。1982 年前，决赛阶段只有 16 支球队 [53]。

简单起见，让我们回到 1978 年，并且想像一种特殊的方法来决定谁是冠军：在第一轮比赛中，所有参赛球队被分为 8 组进行比赛；比赛产生 8 支获胜球队，其余 8 支被淘汰。接下来，在第二轮比赛中，8 支球队被分成 4 组。比赛产生 4 支获胜球队；然后这 4 支球队分成两对，比赛产生最终的两支球队争夺冠军。

经过 4 轮比赛，冠军就可产生。总共的比赛场次为：$8+4+2+1=15$。但是我们并不满足仅仅知道谁是冠军，我们还想知道哪支球队是亚军。

有人会问最后一场比赛中被冠军击败的队伍不是亚军么？在真实的世界杯中，的确如此。但是这个规则在某种程度上并不公平。

我们常常听说过“死亡之组”，假设巴西队一开始就和荷兰队进行比赛。虽然它们两个都是强队，但是必须有一支在一上来就被淘汰。这支被淘汰的球队，很可能会打败除冠军外的其他所有球队。图9.6描述了这一情况。

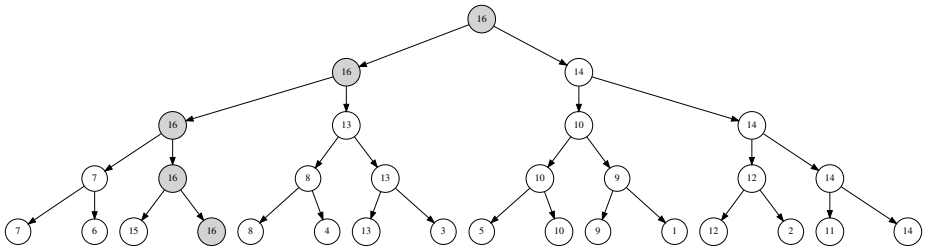


图 9.6: 元素 15 在第一轮就被淘汰

假设每支队伍有一个代表其实力的数字。数字越大，实力越强。假设数字较大的队永远会战胜数字较小的队。虽然现实中不会这样，但是我们可以由此简化模型，给

出一个锦标赛淘汰法的实现。代表冠军的数字为 16, 根据假设的规则，数字 14 不是亚军，而是在第一轮就被淘汰的 15.

我们需要找到一种快速的方法在锦标赛树中找到第二个最大值。此后，我们只要不断重复这一方法，逐一找出第三大，第四大……就可以完成基于选择的排序。

一种办法是，把冠军的数字变成一个很小的值（例如 $-\infty$），这样以后它就不会被选中，这样第二名就会成为新的冠军。假设有 2^m 支球队，其中 m 是某个自然数，仍然需要 $2^{m-1} + 2^{m-2} + ... + 2 + 1 = 2^m - 1$ 次比较才能产生新的冠军，这和第一次寻找冠军花费的代价相同。

实际上，我们无需再进行自底向上的比较。锦标赛树中保存了足够的顺序信息。实力第二强的队，一定在某个时刻被冠军击败，否则它就会是最终的冠军。因此我们可以从锦标赛树的根节点出发，沿着产生冠军的路径向叶子方向遍历，在这条路径上寻找第二强的队。

图9.6中，这条路径被标记为灰色，需要检查的元素包括 $\{14, 13, 7, 15\}$，根据这一思想，我们将算法调整如下：

1. 从待排序元素构建一棵锦标赛树，冠军（最大值）位于树根；
2. 取出树根，自顶向下沿着冠军路径将最大值替换为 $-\infty$；
3. 自底向上沿着刚才的路径回溯，找出新的冠军，并将其置于树根；
4. 重复步骤 2，直到所有的元素都被取出。

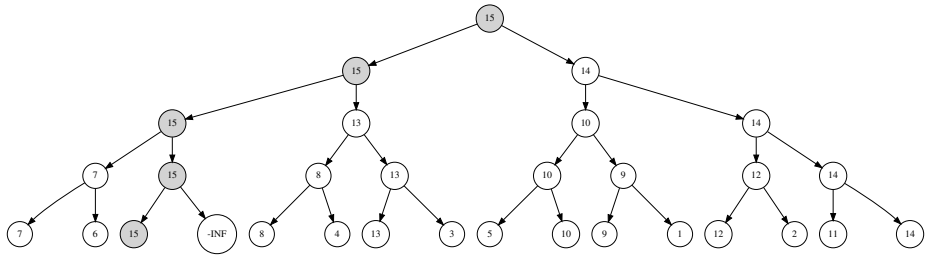
图9.7给出了这一排序的前几个步骤。

我们可以复用二叉树的定义来表示锦标赛树，为了自底向上回溯，每个节点需要同时指向它的父节点。

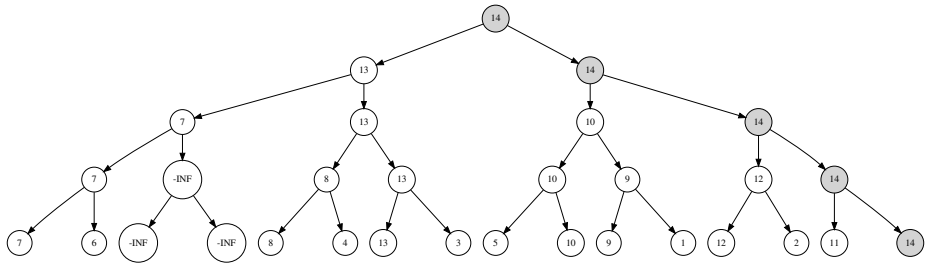
```
struct Node {
    Key key;
    struct Node *left, *right, *parent;
};
```

假设待排序的元素有 2^m 个，其中 m 为某个自然数，为了构造锦标赛树，我们先将每个元素放入一个叶子节点中，这样就得到了一组二叉树的列表。每次从列表中取出两棵树，比较它们根节点的值，然后构造一棵较大的二叉树，树根为其中较大的值，参与比较的两棵树分别作为左右子树。重复这一步骤可以得到一组新的树，其中每棵树的高度增加了 1。这样一轮过后，新得到的树的数目减半。持续同样的操作最终得到一棵锦标赛树。

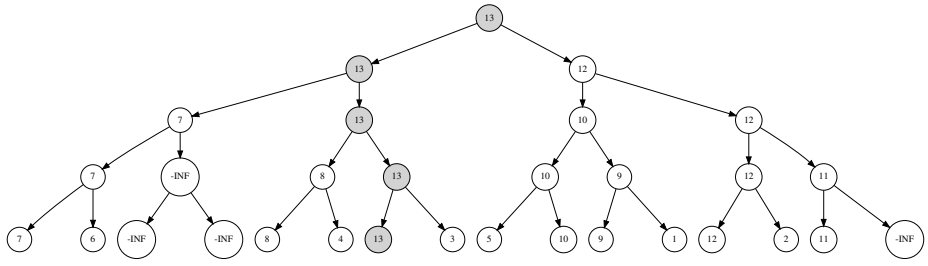
```
1: function BUILD-TREE(A)
2:     T ← φ
3:     for each x ∈ A do
4:         t ← CREATE-NODE
5:         KEY(t) ← x
```



取出 16，将其替换为 $-\infty$，15 上升为新的根



取出 15，将其替换为 $-\infty$，14 上升为新的根



取出 14，将其替换为 $-\infty$，13 上升为新的根

图 9.7: 锦标赛树排序的前几步

```
        APPEND(T, t)
    while |T| > 1 do
        T' ← φ
        for every t1, t2 ∈ T do
            t ← CREATE-NODE
            KEY(t) ← MAX(KEY(t1), KEY(t2))
            LEFT(t) ← t1
            RIGHT(t) ← t2
            PARENT(t1) ← t
            PARENT(t2) ← t
            APPEND(T', t)
        T ← T'
    return T[1]
```

设列表 A 的长度为 n，算法先遍历列表构建树，这一步耗时是线性时间 n，然后它不断取出一对树进行比较，这一步的时间正比于 $n + \frac{n}{2} + \frac{n}{4} + ... + 2 = 2n$。因此整体性能为 $O(n)$。

下面的 C 程序实现了锦标赛树的构造算法。

```
struct Node* build(const Key* xs, int n) {
    int i;
    struct Node *t, **ts = (struct Node**) malloc(sizeof(struct Node*) * n);
    for (i = 0; i < n; ++i)
        ts[i] = leaf(xs[i]);
    for (; n > 1; n /= 2)
        for (i = 0; i < n; i += 2)
            ts[i/2] = branch(max(ts[i]→key, ts[i+1]→key), ts[i], ts[i+1]);
    t = ts[0];
    free(ts);
    return t;
}
```

其中 key 的类型预先定义好，例如：

```
typedef int Key;
```

函数 `leaf(x)` 从值 `x` 构建一个叶子节点。节点的左右分支和父节点都为空。函数 `branch(key, left, right)` 创建一个分支节点，并且令左右子节点的父指针指向新建的分支节点。我们这里省略了 `leaf` 和 `branch` 函数的实现细节。读者可以作为练习，尝试实现上述算法。

某些编程环境，例如 Python 提供了一次迭代两个元素的工具，例如：

```
for x, y in zip(*[iter(ts)]*2):
```

我们略过这些语言细节，读者可以参考本书附带的代码。

每次取出锦标赛树的根节点后，我们自顶向下将其替换为 $-\infty$，然后在通过父指针向上回溯，找出新的最大值。

```
1: function EXTRACT-MAX(T)
2:     m ← KEY(T)
3:     KEY(T) ← −∞
4:     while ¬ LEAF?(T) do                          ▷ 自顶向下一轮
5:         if KEY(LEFT(T)) = m then
6:             T ← LEFT(T)
7:         else
8:             T ← RIGHT(T)
9:         KEY(T) ← −∞
10:    while PARENT(T) ≠ φ do                       ▷ 自底向上一轮
11:        T ← PARENT(T)
12:        KEY(T) ← MAX(KEY(LEFT(T)), KEY(RIGHT(T)))
13:    return m
```

这一算法返回最大元素，并更改锦标赛树。在真实的编程环境中，由于有限的字长，我们无法使用真正的 $-\infty$。通常使用一个相对大的负数，它比锦标赛树中的任何元素都小。例如，若所有的元素都大于-65535，我们可以定义负无穷为：

```
#define N_INF -65535
```

下面的 C 例子程序实现了这一算法。

```
Key pop(struct Node* t) {
    Key x = t→key;
    t→key = N_INF;
    while (!isleaf(t)) {
        t = t→left→key == x ? t→left : t→right;
        t→key = N_INF;
    }
    while (t→parent) {
        t = t→parent;
        t→key = max(t→left→key, t→right→key);
    }
    return x;
}
```

EXTRACT-MAX 的行为和某些数据结构的弹出操作非常类似。例如队列和堆，因此我们在上述代码中将其命名为 pop。

EXTRACT-MAX 上下处理树两遍，首先自顶向下一遍，接着自底向上沿着“冠军之路”一遍。由于锦标赛树是平衡的，路径的长度，也就是树的高度为 $O(\lg n)$，其中 n 是待排序元素的数目（也就是叶子节点的数目）。因此算法的性能为 $O(\lg n)$。

为了实现锦标赛淘汰排序算法，我们先从待排序元素构造一棵锦标赛树，然后不断取出最大值。如果我们希望按照单调递增的顺序排序，我们将第一个取出的元素放

在最右侧，然后将后继取出的元素依次向左侧放；否则，如果是降序排序，我们不断将取出的元素追加到结果的末尾。下面的算法按照升序进行排序。

```
1: procedure SORT(A)
2:     T ← BUILD-TREE(A)
3:     for i ← |A| down to 1 do
4:         A[i] ← EXTRACT-MAX(T)
```

下面的 C 例子程序实现了上述排序算法。

```
void tsort(Key* xs, int n) {
    struct Node* t = build(xs, n);
    while(n)
        xs[--n] = pop(t);
    release(t);
}
```

算法首先使用 $O(n)$ 时间构建一棵锦标赛树，然后执行 n 次弹出操作，逐一从树中取出剩余元素的最大值。因为每次弹出操作的性能为 $O(\lg n)$，所以锦标赛淘汰排序算法的总体性能为 $O(n \lg n)$。

锦标赛淘汰法的细节改进

锦标赛淘汰法也可以用纯函数式的方式实现。我们会看到弹出操作中的两遍处理过程（第一遍自顶向下将冠军替换为 $-\infty$；第二遍自底向上查找新的冠军）可以通过递归合并起来。于是不再需要存储父节点的引用。我们可以复用函数式的二叉树定义，如下面的例子 Haskell 代码所示：

```
data Tr a = Empty | Br (Tr a) a (Tr a)
```

一棵二叉树或者为空，或者为一个分支节点，包含一个 key 和左右子树。每棵子树都是一棵二叉树。

此前，我们使用一个较大的负整数来表示 $-\infty$。但是这个方法是临时性的，有诸多不便。某些编程环境支持代数类型，这样就可以明确定义负无穷。例如下面的 Haskell 程序建立了无穷的定义[3].。

```
data Infinite a = NegInf | Only a | Inf deriving (Eq, Ord)
```

接下来的部分，我们使用 $min()$ 函数来决定比赛的胜者，相应的锦标赛树选择最小的元素作为冠军。

记函数 $key(T)$ 返回树 T 根节点的 key。函数 $wrap(x)$ 将元素 x 装入一个叶子节点。函数 $tree(l, k, r)$ 构造一个分支节点。其中 k 是 key，l 和 r 分别是左右分支。

[3]如果希望直接使用默认的 `Ord` 来比较大小，则需要按照负无穷、普通数字和正无穷的顺序来声明。当然，也可以将我们的类型声明为 `Ord` 的一个 instance，然后给出大小比较的规则。这些是语言特有的性质，超出了本书的范围。读者可以参考其他 Haskell 资料

在淘汰过程中，我们比较两棵树，选择较小的 key 作为新节点的 key，进行比较的两棵树作为左右子树。

$$branch(T_1, T_2) = tree(T_1, min(key(T_1), key(T_2)), T_2) \tag{9.12}$$

对应的 Haskell 例子代码为：

```
branch t1 t2 = Br t1 (min (key t1) (key t2)) t2
```

此前的锦标赛排序算法有一个限制。它要求待排序的元素个数必须是 2^m，否则我们无法构造一棵完全二叉树。现在考虑如何克服这一问题。每次我们都选出两棵树，比较并且选择较大的。如果树的总数目为偶数，我们总能不断选出两棵。在真正的足球比赛中，如果某支球队因故缺席了比赛（例如航班延误），则会有一支球队没有对手。可以规定这支球队为胜者，直接进入接下来的比赛。我们完全可以使用类似的方法。

首先将每个元素都装入叶子节点，然后开始构造锦标赛树。

$$build(L) = build'(\{wrap(x) | x \in L\}) \tag{9.13}$$

函数 $build'(\mathbb{T})$ 中，如果列表 $\mathbb{T}$ 中仅有一棵树，则此树就是最终结果。否则，它将每两棵树分成一组，然后决定胜者。如果有奇数棵树，就规定最后一棵树为胜者，可以进入下一轮比赛。然后我们递归调用这一构造算法。

$$build'(\mathbb{T}) = \begin{cases} \mathbb{T} & : & |\mathbb{T}| \leq 1 \\ build'(pair(\mathbb{T})) & : & otherwise \end{cases} \tag{9.14}$$

这一算法还能处理另外一种特殊情况：如果待排序的列表为空，则结果也为空。

如果列表中至少有两棵树，记 $\mathbb{T} = \{T_1, T_2, ...\}$，而 $\mathbb{T}'$ 表示除最初两棵树外的剩余树。函数 $pair(\mathbb{T})$ 定义如下：

$$pair(\mathbb{T}) = \begin{cases} \{branch(T_1, T_2)\} \cup pair(\mathbb{T}') & : & |\mathbb{T}| \geq 2 \\ \mathbb{T} & : & otherwise \end{cases} \tag{9.15}$$

下面的 Haskell 例子代码给出了构造锦标赛树的完整程序。

```
fromList :: (Ord a) ⇒ [a] → Tr (Infinite a)
fromList = build ∘ (map wrap) where
  build [] = Empty
  build [t] = t
  build ts = build $ pair ts
  pair (t1:t2:ts) = (branch t1 t2):pair ts
  pair ts = ts
```

为了从锦标赛树中取得冠军（最小元素），我们检查左右子树，看哪一棵子树的 key 和根节点的 key 相等。然后递归地从子树中取出冠军直到到达叶子节点。记 T 的

左子树为 L，右子树为 R，K 为 key，弹出算法可以定义如下：

$$pop(T) = \begin{cases} tree(\phi, \infty, \phi) & : & L = \phi \wedge R = \phi \\ tree(L', min(key(L'), key(R)), R) & : & K = key(L), L' = pop(L) \\ tree(L, min(key(L), key(R')), R') & : & K = key(R), R' = pop(R) \end{cases} \tag{9.16}$$

下面的 Haskell 例子代码实现了弹出算法。

```
pop (Br Empty _ Empty) = Br Empty Inf Empty
pop (Br l k r) | k == key l = let l' = pop l in Br l' (min (key l') (key r)) r
               | k == key r = let r' = pop r in Br l (min (key l) (key r')) r'
```

注意这一算法仅仅将冠军元素删除而没有返回，因此有必要定义另外一个函数从根节点提取冠军元素。

$$top(T) = key(T) \tag{9.17}$$

使用这些定义好的函数，锦标赛淘汰排序法可以形式化为下面的等式：

$$sort(L) = sort'(build(L)) \tag{9.18}$$

其中 $sort'(T)$ 不断从锦标赛树中弹出最小的元素：

$$sort'(T) = \begin{cases} \phi & : & T = \phi \vee key(T) = \infty \\ \{top(T)\} \cup sort'(pop(T)) & : & otherwise \end{cases} \tag{9.19}$$

下面的 Haskell 例子程序实现了完整的锦标赛淘汰排序算法。

```
top = only ∘ key

tsort :: (Ord a) ⇒ [a] → [a]
tsort = sort' ∘ fromList where
    sort' Empty = []
    sort' (Br _ Inf _) = []
    sort' t = (top t) : (sort' $ pop t)
```

其中用以支持无穷类型的辅助函数 `only`、`key` 和 `wrap` 定义如下：

```
only (Only x) = x
key (Br _ k _ ) = k
wrap x = Br Empty (Only x) Empty
```

练习 9.3

- 实现命令式锦标赛淘汰法中的辅助函数 `leaf()`、`branch`、`max()`、`isleaf()` 和 `release()`。
- 在一门支持垃圾回收（GC）的语言中实现命令式的锦标赛淘汰法排序程序。

- 为什么我们的锦标赛树淘汰法排序程序可以处理重复元素（元素的值相等）？如果相等元素的顺序经过排序后保持不变，我们称之为稳定排序。锦标赛树淘汰法排序是稳定排序么？
- 设计一个命令式的锦标赛淘汰排序算法，满足下面的条件：
  - 可以处理任意数目的元素；
  - 不使用硬编码（hard code）的大负数，可以处理任意值的元素。
- 比较锦标赛树淘汰算法和二叉搜索树排序算法，分析它们的时间和空间效率。
- 比较堆排序算法和二叉搜索树排序算法，分析它们的时间和空间效率。

9.4.2 使用堆排序进行最后的改进

通过使用锦标赛树淘汰法，我们将基于选择的排序算法性能提高到 $O(n \lg n)$。这已经达到了基于比较的排序算法的上限 [51]。但是，这里仍然有提高的空间。排序完成后，锦标赛树的所有节点都变成了负无穷，这棵完全二叉树不再含有任何有用的信息，但它却占据了很大空间。有没有办法在弹出后释放节点呢？

另外我们可以观察到，如果待排序的元素有 n 个，我们实际上使用了 $2n$ 个节点。其中有 n 个叶子和 n 个分支。有没有办法能节约一半空间呢？

如果我们认为根节点的 key 为无穷，则树为空，那么上一节最后给出的公式9.19就可以进一步概括为更加通用的形式：

$$sort'(T) = \begin{cases} \phi & : & T = \phi \\ \{top(T)\} \cup sort'(pop(T)) & : & otherwise \end{cases} \tag{9.20}$$

这和我们在上一章堆排序给出的公式完全一样。堆总是在顶部保存最小（或最大）值，并且提供了快速的弹出操作。使用数组的 binary 堆实际上将树结构“编码”成数组的索引，因此除了 n 个单元外，无需任何额外的空间。函数式的堆，如左偏堆和 splay 堆也只需要 n 个节点。我们将在下一章介绍更多种类的堆，它们在许多情况下都有很好的性能。

9.5 小结

本章我们介绍了选择排序的进化过程。选择排序简单、直观，经常被用来教授编程中的多重循环。它的结构虽然简单，但是性能却是平方级别的。本章中，我们看到，选择排序不仅可以通过细微调整加以改进，而且还可以通过改变底层的数据结构，进化到锦标赛淘汰排序和堆排序，从而在本质上得到性能的提升。

第十章　二项式堆，斐波那契堆和配对堆

10.1　简介

在此前的章节中，我们看到通用的堆可以由各种不同的数据结构来实现。我们介绍了用二叉树实现的各种堆。

将二叉树进行扩展可以得到 K 叉树 [54]。本章中，我们首先介绍二项式堆，它由 K 叉树的森林组成，可以在常数时间获取堆顶元素，其他操作的性能都可以达到 $O(\lg n)$。

如果延迟某些二项式堆的操作，就可以得到斐波那契堆。我们此前介绍的 binary 堆都最少需要 $O(\lg n)$ 时间来实现合并，而斐波那契堆可以将合并操作提高到常数时间 $O(1)$。这对于图算法很重要。实际上，斐波那契堆的大部分操作的分摊性能都是常数时间 $O(1)$ 的，只有弹出操作为 $O(\lg n)$。

最后，我们将介绍配对（pairing）堆。它在实际中拥有最好的性能。但是迄今为止，它的性能还是个猜想，没有得到最终的证明。

10.2　二项式堆

本节我们介绍二项式堆。二项式堆由一组 K 叉树的森林组成，它的名称来自于数学中的牛顿二项式展开。在二项式堆的森林中，树木的大小为二项式展开中的各项系数。

10.2.1　定义

二项式堆比大部分的 binary 堆都复杂，但是合并操作的性能很好，可以达到 $O(\lg n)$。一个二项式堆包含一组二项式树。

二项式树

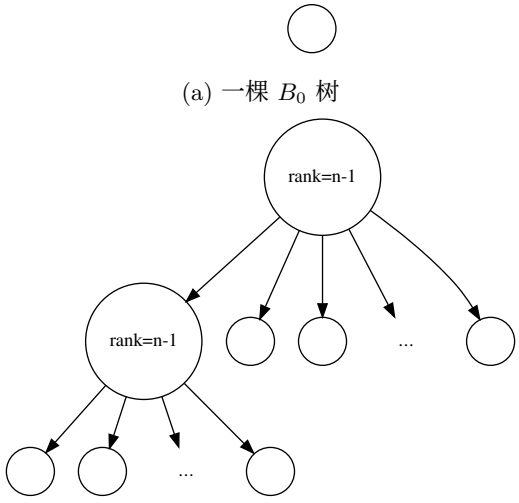
为了了解“二项式树”名字的由来，我们先看一下著名的帕斯卡三角形（中国称为“贾宪”三角形以纪念古代中国的数学家贾宪（1010-1070））[55]。

```
    1
   1 1
  1 2 1
 1 3 3 1
1 4 6 4 1
...
```

每行的数字都是二项式系数。有很多方法可以获得一系列二项式系数，其中一种是使用递归组合。同样，二项式树也可以用类似的方法定义：

- Rank 为 0 的二项式树只有一个根节点；
- Rank 为 n 的二项式树包含两棵 Rank 为 $n-1$ 的二项式树，两棵树中，根节点元素较大的一棵被链接为另一棵最左侧的子树。

我们记 Rank 为 0 的二项式树为 B_0，Rank 为 n 的二项式树为 B_n。
图10.1描述了 B_0，以及如何将两棵 B_{n-1} 树链接成 B_n。



(a) 一棵 B_0 树

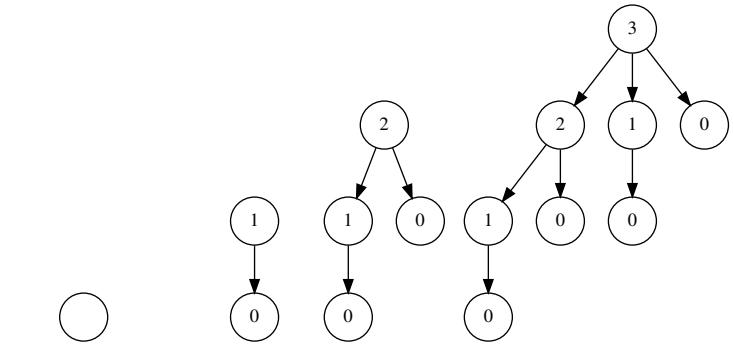
(b) 将两棵 B_{n-1} 树链接成一棵 B_n 树

图 10.1: 二项式树的递归定义

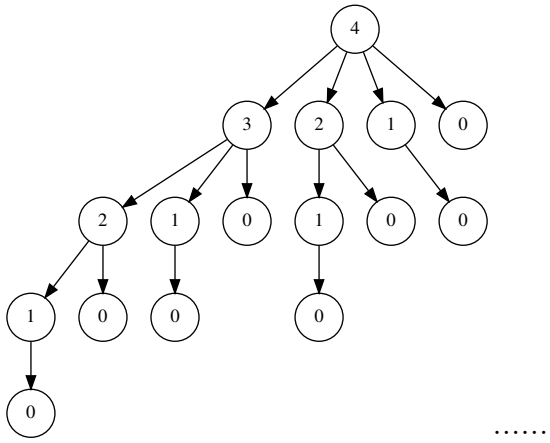
使用这一递归定义，我们可以画出 Rank 分别为 0、1、2……的各个二项式树形式，如图10.2所示。

观察这些二项式树可以发现一些有趣的性质。对于任意 Rank 为 n 的二项式树，每行的节点数目恰好是二项式系数。

例如 Rank 为 4 的二项式树，第一层根有一个节点；第二层有 4 个节点；第三层有 6 个节点；第四层有 4 个节点；第五层有 1 个节点。它们恰好是帕斯卡三角形的第 5 行：1、4、6、4、1。这就是二项式树名字的由来。



(a) B_0 树 (b) B_1 树 (c) B_2 树 (d) B_3 树



(e) B_4 树

图 10.2: Rank 为 0、1、2、3、4……的二项式树

另外一个有趣的性质是，一棵 Rank 为 n 的二项式树中的总节点数为 2^n。我们可以直接用二项式定理或者用递归定义来证明它。

二项式堆

利用二项式树，我们可以给出二项式堆的定义。一个二项式堆是一组二项式树（或称为一个二项式树森林），它满足如下性质：

- 堆中的每棵树都满足堆性质，即任意节点的 key 都大于等于父节点。这里使用了最小堆。也可以使用最大堆，但需要将条件变为“小于等于”。简单起见，本章仅讨论最小堆。所有内容都可以通过改变比较条件，变为最大堆。
- 堆中最多有一棵二项式树的 Rank 为 r。换言之，堆中任何两棵二项式树的 Rank 都不同。

这一定义直接导致了一个重要的结果。对于含有 n 个元素的二项式堆，如果将 n 转换为二进制数 $a_0, a_1, a_2, ..., a_m$，其中 a_0 是最低位（LSB），a_m 是最高位（MSB）。对于任意 $0 \leq i \leq m$，若 $a_i = 0$，则堆中不存在 Rank 为 i 的二项式树；若 $a_i = 1$，则堆中一定含有一棵 Rank 为 i 的二项式树。

例如，某个二项式堆含有 5 个元素，因为 5 的二进制为“(LSB) 101(MSB)”，因此这个堆中含有两棵二项式树，一棵的 Rank 为 0，另外一棵的 Rank 为 2。

图10.3所示的二项式堆含有 19 个节点，因为 19 的二进制为“(LSB)11001(MSB)”，因此含有一棵 B_0 树、一棵 B_1 树和一棵 B_4 树。

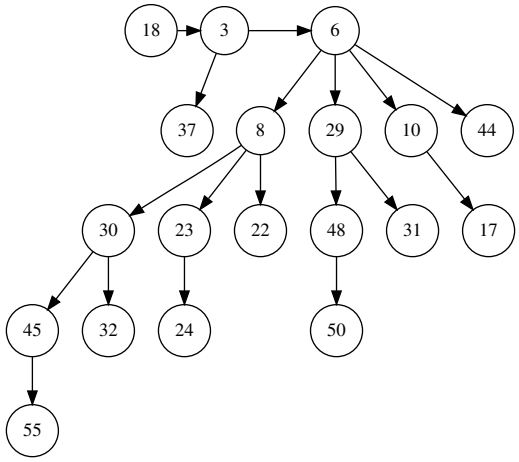


图 10.3: 含有 19 个元素的二项式堆

数据布局

在命令式环境中，有两种方法可以定义 K 叉树。一种是使用“左侧孩子，右侧兄弟”（left-child, right-sibling）的方法 [4]。好处是这种定义和典型的二叉树结构一

致。每个节点包含两个字段（field），左侧字段和右侧字段。我们使用左侧的字段指向节点的第一棵子树，用右侧字段指向此节点的兄弟节点。所有的兄弟节点被串联成一个单向链表。如图10.4所示。

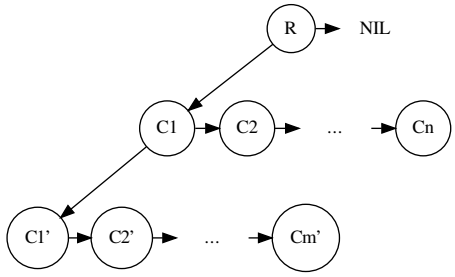


图 10.4: “左侧孩子，右侧兄弟”的例子。R 为根节点，它没有兄弟节点，因此它的右侧指向空 NIL，$C_1, C_2, ..., C_n$ 为 R 的子节点。R 的左侧链接到 C_1，其他 C_1 的兄弟节点依次向右链接起来。$C'_2, ..., C'_m$ 为 C_1 的子节点。

另外一种方法是使用容器类数据结构，如使用数组或者链表来存储节点的所有子树。

因为二项式树的 Rank 很重要，我们将其也定义为一个字段。

下面的 Python 例子程序，使用“左侧孩子，右侧兄弟”方法，定义了二项式树：

```
class BinomialTree:
    def __init__(self, x = None):
        self.rank = 0
        self.key = x
        self.parent = None
        self.child = None
        self.sibling = None
```

当使用一个元素 key 来构造树时，我们创建一个叶子节点，其 Rank 为 0，其他字段都为空。

下面的例子程序使用列表来存储子节点，相应的定义如下：

```
class BinomialTree:
    def __init__(self, x = None):
        self.rank = 0
        self.key = x
        self.parent = None
        self.children = []
```

在纯函数式环境中，例如 Haskell，我们可以将二项式树定义如下：

```
data BiTree a = Node { rank :: Int
                     , root :: a
                     , children :: [BiTree a]}
```

而二项式堆被定义为二项式树的列表（森林），其中的树按照 Rank 单调递增排序。并且符合一个额外的限制：没有任何两棵树的 Rank 相等。

```
type BiHeap a = [BiTree a]
```

10.2.2 基本的堆操作

本节我们介绍二项式堆的基本的操作，包括树的链接，插入新元素，堆的合并，以及访问和弹出堆顶元素。

树的链接

为了实现基本的堆操作如弹出、插入，我们需要先实现将两棵 Rank 一样的树链接成一棵较大的树。根据二项式树的定义，以及根必须保存最小值的堆性质，需要先比较两棵树的根节点，选取较小的一个作为新的根，然后将另一棵树插入到其他子树的前面，如图10.5所示。设函数 $Key(T)$、$Children(T)$ 和 $Rank(T)$ 分别访问树的 key，子树和 Rank。

$$link(T_1, T_2) = \begin{cases} node(r+1, x, \{T_2\} \cup C_1) & : \quad x < y \\ node(r+1, y, \{T_1\} \cup C_2) & : \quad otherwise \end{cases} \tag{10.1}$$

其中

$$\begin{aligned} x &= Key(T_1) \\ y &= Key(T_2) \\ r &= Rank(T_1) = Rank(T_2) \\ C_1 &= Children(T_1) \\ C_2 &= Children(T_2) \end{aligned}$$

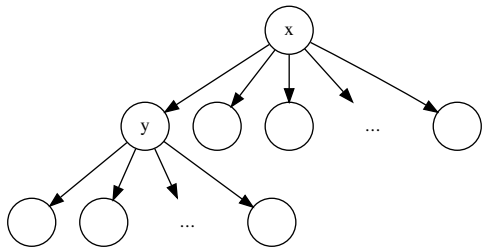


图 10.5: 如果 $x < y$，将 y 作为 x 的第一个子树插入。

如果 $\cup$ 是一个常数时间操作，则链接操作的性能也为 $O(1)$。下面的 Haskell 例子程序实现了链接。

```
link t1@(Node r x c1) t2@(Node _ y c2) =
    if x<y then Node (r+1) x (t2:c1)
    else Node (r+1) y (t1:c2)
```

链接操作也可以用命令式的方式实现。如果使用“左侧孩子，右侧兄弟”的布局，只需要将 key 较大的树链接到另一棵树的左侧字段，然后将子树链接到 key 较大的树的右侧字段。如图10.6所示。

1: **function** LINK(T_1, T_2)
2: **if** KEY(T_2) $<$ KEY(T_1) **then**
3: Exchange $T_1 \leftrightarrow T_2$
4: SIBLING(T_2) $\leftarrow$ CHILD(T_1)
5: CHILD(T_1) $\leftarrow T_2$
6: PARENT(T_2) $\leftarrow T_1$
7: RANK(T_1) $\leftarrow$ RANK(T_1) $+ 1$
8: **return** T_1

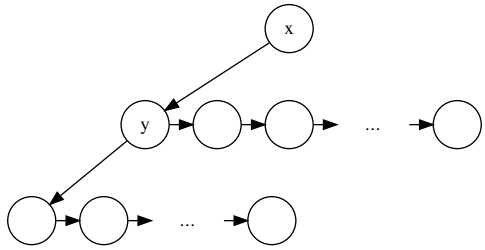


图 10.6: 若 $x < y$，将 y 链接到 x 的左侧，将 x 的子树链接到 y 的右侧。

如果使用容器来存储节点的子树，相应的算法如下：

1: **function** LINK'(T_1, T_2)
2: **if** KEY(T_2) $<$ KEY(T_1) **then**
3: Exchange $T_1 \leftrightarrow T_2$
4: PARENT(T_2) $\leftarrow T_1$
5: INSERT-BEFORE(CHILDREN(T_1), T_2)
6: RANK(T_1) $\leftarrow$ RANK(T_1) $+ 1$
7: **return** T_1

上述两个算法都易于实现。这里我们给出了 LINK' 的 Python 例子程序。

```
def link(t1, t2):
    if t2.key < t1.key:
        (t1, t2) = (t2, t1)
    t2.parent = t1
    t1.children.insert(0, t2)
    t1.rank = t1.rank + 1
    return t1
```

练习 10.1

选择一门语言，用“左侧孩子，右侧兄弟”的布局实现二项式树的链接程序。

如果使用“左侧孩子，右侧兄弟”的实现，链接可以在常数时间完成，但是如果使用容器来存储子树，则性能依赖于容器的具体实现。如果容器是基于数组的，链接操作的时间和子树的数目成正比；而如果使用链表，则为常数时间。本章中，我们认为这一时间是常数的。

插入新元素（push）

如果森林中的二项式树的 Rank 是单调增的，通过使用 $link$ 函数，我们可以定义一个辅助函数用于向堆中插入一棵新树。这棵新树的 Rank 不大于森林中的任何树。

记非空的堆为 $H = \{T_1, T_2, ..., T_n\}$，我们定义：

$$insertT(H, T) = \begin{cases} \{T\} & : & H = \phi \\ \{T\} \cup H & : & Rank(T) < Rank(T_1) \\ insertT(H', link(T, T_1)) & : & otherwise \end{cases} \tag{10.2}$$

其中

$$H' = \{T_2, T_3, ..., T_n\}$$

如果堆为空，则新树为森林中唯一的一棵；否则，我们比较新树和森林中第一棵树的 Rank，如果相等，就将它们链接成一棵更大的树（Rank 加 1），然后递归地插入到森林中；如果不等，根据限制条件，新树的 Rank 必然是最小的，我们将它插入到森林中所有树的前面。

根据此前给出的二项式堆的性质，如果元素总数为 n，森林中最多有 $O(\lg n)$ 棵二项式树。函数 $insertT$ 最多执行 $O(\lg n)$ 次常数时间的链接操作。因此 $insertT$ 的性能为 $O(\lg n)$[1]。

相应的 Haskell 例子程序如下：

```
insertTree [] t = [t]
insertTree ts@(t':ts') t = if rank t < rank t' then t:ts
                           else insertTree ts' (link t t')
```

使用这一辅助函数，我们可以实现堆的插入算法。先将待插入元素装入一个叶子节点，然后将它插入到二项式堆中。

$$insert(H, x) = insertT(H, node(0, x, \phi)) \tag{10.3}$$

我们可以连续将若干元素通过 folding 插入到堆中，下面的 Haskell 例子程序定义了一个辅助函数 `fromList`。

[1] 观察这一操作和两个二进制数的加法，它们有很多相似性。可以引出一个有趣的题目：“numeric representation” [3]。

```
fromList = foldl insert []
```

因为将元素放入叶子节点只需要常数时间，主要的工作由 $insertT$ 完成，所以二项式堆的插入操作性能为 $O(\lg n)$。

插入算法也可以用命令式的方式定义。

Algorithm 1 使用“左侧孩子，右侧兄弟”的实现插入一棵新树

```
1: function INSERT-TREE(H, T)
2:     while H ≠ φ∧ RANK(HEAD(H)) = RANK(T) do
3:         (T_1, H) ← EXTRACT-HEAD(H)
4:         T ← LINK(T, T_1)
5:     SIBLING(T) ← H
6:     return T
```

如果 Rank 相等，算法 Algorithm 1不断将堆中第一棵树和待插入的树链接到一起。此后，它将剩余的树作为兄弟链接到末尾，然后将新的链表返回。

如果使用容器来存储子树，则算法定义为 Algorithm 2。

Algorithm 2 插入一棵新树，使用容器来存储子树

```
1: function INSERT-TREE′(H, T)
2:     while H ≠ φ∧ RANK(H[0]) = RANK(T) do
3:         T_1 ← POP(H)
4:         T ← LINK(T, T_1)
5:     HEAD-INSERT(H, T)
6:     return H
```

其中函数 POP 将森林中的第一棵树 $T_1 = H[0]$ 取出。函数 HEAD-INSERT 将新树添加到堆中所有树的前面。

使用 INSERT-TREE 或 INSERT-TREE′ 中的任何一个，都可以实现二项式堆的插入算法。

Algorithm 3 命令式插入算法

```
1: function INSERT(H, x)
2:     return INSERT-TREE(H, NODE(0, x, φ))
```

下面的 Python 程序使用内置的列表实现了上述算法，使用“左侧孩子，右侧兄弟”的实现留给读者作为练习。

```
def insert_tree(ts, t):
    while ts ≠[] and t.rank == ts[0].rank:
        t = link(t, ts.pop(0))
```

```
    ts.insert(0, t)
    return ts

def insert(h, x):
    return insert_tree(h, BinomialTree(x))
```

练习 10.2

选择一门命令式编程语言，利用“左侧孩子，右侧兄弟”的布局实现二项式堆的插入算法。

堆合并

合并两个二项式堆等价于合并两个二项式树的森林。根据定义，合并后的森林中没有 Rank 相同的树，并且树按照 Rank 单调递增的顺序排列。合并过程的思想和归并排序类似。在每次迭代中，我们从两个森林中各取出第一棵树，比较它们的 Rank，将较小的一棵树放入结果堆中；如果 Rank 相等，我们将它们链接起来成为一棵新树，然后递归将它插入到剩余树的合并结果中。

图10.7描述了这一算法。它和 [4] 中介绍的实现并不相同。

可以将合并算法形式化为一个函数。若两个堆不为空，分别记它们为：$H_1 = \{T_1, T_2, ...\}$、$H_2 = \{T_1', T_2', ...\}$。并且令 $H_1' = \{T_2, T_3, ...\}$、$H_2' = \{T_2', T_3', ...\}$。

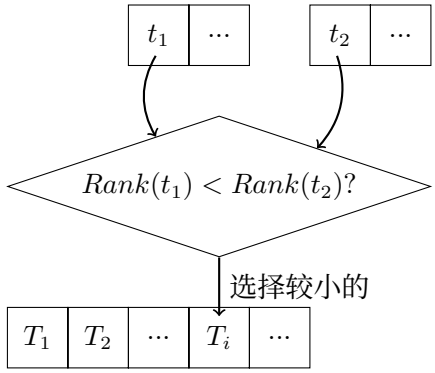
$$merge(H_1, H_2) = \begin{cases} H_1 & : & H_2 = \phi \\ H_2 & : & H_1 = \phi \\ \{T_1\} \cup merge(H_1', H_2) & : & Rank(T_1) < Rank(T_1') \\ \{T_1'\} \cup merge(H_1, H_2') & : & Rank(T_1) > Rank(T_1') \\ insertT(merge(H_1', H_2'), link(T_1, T_1')) & : & otherwise \end{cases} \tag{10.4}$$

为了分析合并操作的性能，设堆 H_1 中有 m_1 棵树，堆 H_2 中有 m_2 棵树。合并后的结果中最多有 $m_1 + m_2$ 棵树。如果没有 Rank 相同的树，则合并操作的时间为 $O(m_1 + m_2)$。如果存在 Rank 相同的树需要链接，最多需要调用 $insertT$ 的次数为 $O(m_1 + m_2)$。考虑 $m_1 = 1 + \lfloor \lg n_1 \rfloor$，$m_2 = 1 + \lfloor \lg n_2 \rfloor$，其中 n_1 和 n_2 是两个堆各自的节点数目，且 $\lfloor \lg n_1 \rfloor + \lfloor \lg n_2 \rfloor \leq 2\lfloor \lg n \rfloor$，其中 $n = n_1 + n_2$ 为总节点数。最终的合并性能为 $O(\lg n)$。

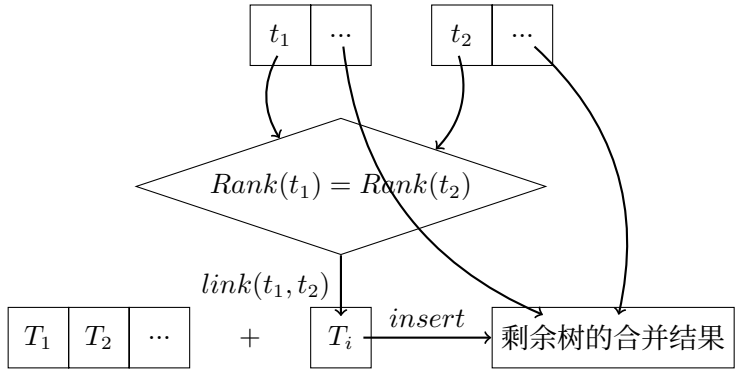
下面的 Haskell 例子程序实现了合并算法。

```
merge ts1 [] = ts1
merge [] ts2 = ts2
merge ts1@(t1:ts1') ts2@(t2:ts2')
    | rank t1 < rank t2 = t1:(merge ts1' ts2)
    | rank t1 > rank t2 = t2:(merge ts1 ts2')
    | otherwise = insertTree (merge ts1' ts2') (link t1 t2)
```

合并算法也可以用命令式的方式实现，如 Algorithm 4。



(a) 选择 Rank 较小的树放入结果中。



(b) 若两棵树的 Rank 相同，将它们链接成一棵新树，然后递归地将它插入的剩余树的合并结果中。

图 10.7: 堆合并

Algorithm 4 命令式合并两个堆

```
1: function MERGE(H_1, H_2)
2:     if H_1 = φ then
3:         return H_2
4:     if H_2 = φ then
5:         return H_1
6:     H ← φ
7:     while H_1 ≠ φ ∧ H_2 ≠ φ do
8:         T ← φ
9:         if RANK(H_1) < RANK(H_2) then
10:            (T, H_1) ← EXTRACT-HEAD(H_1)
11:        else if RANK(H_2) < RANK(H_1) then
12:            (T, H_2) ← EXTRACT-HEAD(H_2)
13:        else                                          ▷ Rank 相等
14:            (T_1, H_1) ← EXTRACT-HEAD(H_1)
15:            (T_2, H_2) ← EXTRACT-HEAD(H_2)
16:            T ← LINK(T_1, T_2)
17:        APPEND-TREE(H, T)
18:    if H_1 ≠ φ then
19:        APPEND-TREES(H, H_1)
20:    if H_2 ≠ φ then
21:        APPEND-TREES(H, H_2)
22:    return H
```

两个堆都含有 Rank 单调递增的二项式树。每次迭代，我们选出 Rank 最小的树并追加到结果堆中。如果两个堆中的第一棵树的 Rank 相等，我们先把它们链接成一棵新树。考虑 APPEND-TREE 过程。根据我们的合并策略，新树的 Rank 不能比结果堆中的任何一棵小，但是它有可能和结果树中的最后一棵相等。链接操作可能会引起这样的情况，因为它会将树的 Rank 增加 1。此时我们需要把新树和结果堆中的最后一棵树链接起来。下面的算法中，函数 $\textsc{Last}(H)$ 给出堆中最后一棵树，函数 $\textsc{Append}(H, T)$ 仅仅将一棵新树追加到森林的末尾。

1: **function** APPEND-TREE(H, T)
2: **if** $H \neq \phi \wedge$ $\textsc{Rank}(T) = \textsc{Rank}(\textsc{Last}(H))$ **then**
3: $\textsc{Last}(H) \leftarrow \textsc{Link}(T, \textsc{Last}(H))$
4: **else**
5: $\textsc{Append}(H, T)$

APPEND-TREES 不断调用上述函数，逐一将一个堆中的树追加到另一堆中。

1: **function** APPEND-TREES(H_1, H_2)
2: **for** each $T \in H_2$ **do**
3: $H_1 \leftarrow$ APPEND-TREE(H_1, T)

下面的 Python 例子程序实现了合并算法。

```
def append_tree(ts, t):
    if ts ≠ [] and ts[-1].rank == t.rank:
        ts[-1] = link(ts[-1], t)
    else:
        ts.append(t)
    return ts

def append_trees(ts1, ts2):
    return reduce(append_tree, ts2, ts1)

def merge(ts1, ts2):
    if ts1 == []:
        return ts2
    if ts2 == []:
        return ts1
    ts = []
    while ts1 ≠ [] and ts2 ≠ []:
        t = None
        if ts1[0].rank < ts2[0].rank:
            t = ts1.pop(0)
        elif ts2[0].rank < ts1[0].rank:
            t = ts2.pop(0)
        else:
            t = link(ts1.pop(0), ts2.pop(0))
        ts = append_tree(ts, t)
    ts = append_trees(ts, ts1)
    ts = append_trees(ts, ts2)
    return ts
```

练习 10.3

例子程序使用了容器来存储子树。选择一门语言，实现“左侧孩子，右侧兄弟”方式堆的合并。

弹出

在二项式堆的森林中，每棵二项式树都符合堆性质，根节点保存了树中的最小元素。但是这些根节点元素间的大小关系是任意的。为了获取堆中的最小元素，我们需要从全部树根中找到最小元素。因为堆中有 $\lg n$ 棵树，所以获取最小值的复杂度为 $O(\lg n)$。

但是弹出操作要求不仅仅找到最小元素（即 top），还需要将其删除并保持堆性质。设构成堆的各个二项式树为 $B_i, B_j, ..., B_p, ..., B_m$，其中 B_k 为 Rank 为 k 的二项式树。设堆中最小元素保存在树 B_p 的根节点。将其删除后，会产生 p 棵子树，它们都是二项式树，Rank 分别为 $p-1, p-2, ..., 0$。

此前我们已经定义了性能为 $O(\lg n)$ 的合并函数。一个思路是将 p 棵子树逆序，这样它们的 Rank 就变为单调递增的，形成一个二项式堆 H_p。剩余的树也构成一个二项式堆，可以表示为 $H' = H - B_p$。将 H_p 和 H' 合并就可以得到弹出操作的最终结果。图10.8描述了这一思路。

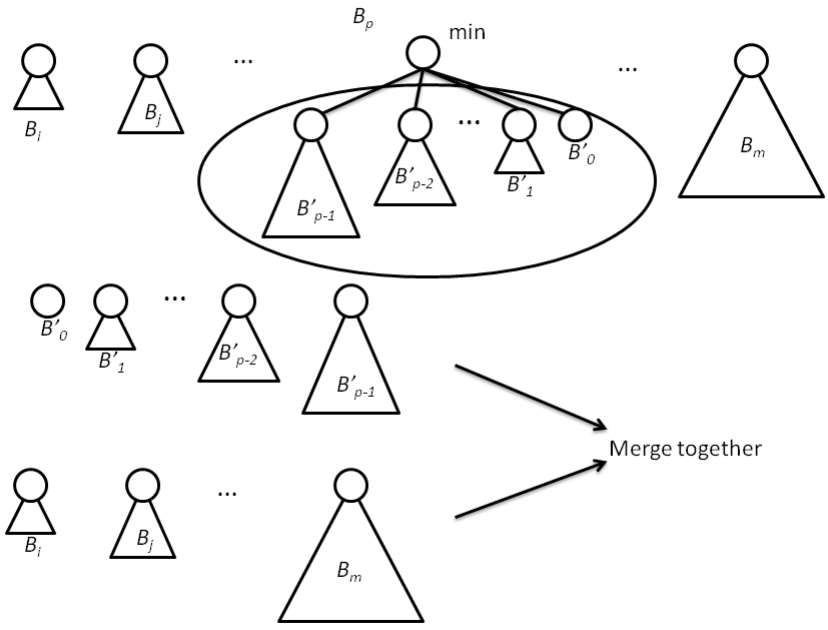


图 10.8: 二项式堆的弹出操作

为了实现弹出算法，我们需要先定义一个函数，可以从森林中取出根节点最小的

树。

$$extractMin(H) = \begin{cases} (T, \phi) & : & \text{H 只含有一个元素，形如}\{T\} \\ (T_1, H') & : & Root(T_1) < Root(T') \\ (T', \{T_1\} \cup H'') & : & otherwise \end{cases} \tag{10.5}$$

其中

$$\begin{aligned} H &= \{T_1, T_2, ...\} && \text{非空的森林;} \\ H' &= \{T_2, T_3, ...\} && \text{去除第一棵树的森林;} \\ (T', H'') &= extractMin(H') \end{aligned}$$

此函数的结果为一对值，第一部分是根节点最小的树，第二部分是森林中剩余的其他树。函数逐一检查并比较森林中的每棵树，因此它的性能为 $O(\lg n)$。

相应的 Haskell 例子程序如下：

```
extractMin [t] = (t, [])
extractMin (t:ts) = if root t < root t' then (t, ts)
                    else (t', t:ts')
    where
      (t', ts') = extractMin ts
```

调用使用这一函数，就可以获得堆顶元素：

```
findMin = root ∘ fst ∘ extractMin
```

当然，也可以仅遍历森林中的所有树，找出最小的根节点而不将树删除。下面的命令式算法使用“左侧孩子，右侧兄弟”的布局实现了最小值的查找。

```
1: function FIND-MINIMUM(H)
2:     T ← HEAD(H)
3:     min ← ∞
4:     while T ≠ φ do
5:         if KEY(T)< min then
6:             min ← KEY(T)
7:         T ← SIBLING(T)
8:     return min
```

如果使用容器来存储子树，就需要在二项式树的列表中寻找根节点最小的一棵。下面的 Python 例子程序给出了这种情况的实现。

```
def find_min(ts):
    min_t = min(ts, key=lambda t: t.key)
    return min_t.key
```

接下来需要使用 $extractMin$ 来定义从堆中删除最小元素的函数。

$$delteMin(H) = merge(reverse(Children(T)), H') \tag{10.6}$$

其中

$$(T, H') = extractMin(H)$$

我们在此略过了相应的 Haskell 例子代码。

为了给出命令式的实现，我们需要额外实现列表反转等操作。我们将其留给读者作为练习。下面的伪代码描述了命令式的弹出算法。

```
1: function EXTRACT-MIN(H)
2:     (T_min, H) ← EXTRACT-MIN-TREE(H)
3:     H ← MERGE(H, REVERSE(CHILDREN(T_min)))
4:     return (KEY(T_min), H)
```

使用弹出操作可以实现堆排序。首先从待排序元素构建一个二项式堆，然后不断从中弹出最小元素直到堆变为空。

$$sort(xs) = heapSort(fromList(xs)) \tag{10.7}$$

其中的 $heapSort$ 实现如下：

$$heapSort(H) = \begin{cases} \phi & : \quad H = \phi \\ \{findMin(H)\} \cup heapSort(deleteMin(H)) & : \quad otherwise \end{cases} \tag{10.8}$$

下面的 Haskell 例子程序实现了堆排序。

```
heapSort = hsort ∘ fromList where
    hsort [] = []
    hsort h = (findMin h):(hsort $ deleteMin h)
```

其中 $fromList$ 函数可以通过 folding 来定义。也可以用命令式的方法实现二项式堆排序，读者可以参考前面 binary 堆的相关章节。

练习 10.4

- 选择一门编程语言，用“左侧孩子，右侧兄弟”的方法实现从二项式堆中取得最小元素的操作。
- 实现命令式的 EXTRACT-MIN-TREE() 算法。
- 使用“左侧孩子，右侧兄弟”的方法，将一棵树的所有子树逆序相当于实现单向链表的反转。选择一门编程语言，实现单向链表的反转。

其他

此前我们给出的二项式堆的插入、合并的性能都是 $O(\lg n)$。这一结论是针对最坏情况的。这些操作的分摊复杂度为 $O(1)$。我们这里略去了分摊复杂度的证明。

10.3 斐波那契堆

“斐波那契堆”的命名很有趣，实际上斐波那契堆的结构和斐波那契数列无关。斐波那契堆的作者 Michael L. Fredman 和 Robert E. Tarjan 在证明这种堆的时间性能时使用了斐波那契数列的性质，于是他们决定给这种堆命名为“斐波那契堆”[4]。

10.3.1 定义

斐波那契堆本质上是一个惰性二项式堆。但是这并不意味着二项式堆在支持惰性求值的环境下（例如 Haskell）自动就成为斐波那契堆。惰性环境仅仅对于实现提供了便利。例如 [56] 中给出了一个简洁的实现。

斐波那契堆在理论上具有良好的性能。除弹出之外所有的操作分摊性能都达到了常数时间 $O(1)$。本节中，我们给出的实现和常见的实现 [4] 有所不同。主要思想来自于 Okasaki 的工作 [57]。

首先我们对比一下二项式堆和斐波那契堆的性能（确切的说，是我们希望斐波那契堆达到的性能目标）。

操作	二项式堆	斐波那契堆
插入	$O(\lg n)$	$O(1)$
合并	$O(\lg n)$	$O(1)$
top	$O(\lg n)$	$O(1)$
弹出	$O(\lg n)$	分摊 $O(\lg n)$

表 10.1: 斐波那契堆的性能目标

在二项式堆中，插入一个新元素时，哪里是瓶颈呢？新元素 x 被放入只有一个叶子节点的树中，然后这棵树被插入到森林中。

在此期间，树按照 Rank 的单调递增顺序插入，如果 Rank 相等，则进行链接，然后再递归，因此性能为 $O(\lg n)$。

使用惰性策略，我们可以将按照 Rank 的顺序插入和链接等操作推迟进行。仅仅将只有一个叶子节点的树放入森林中。这样带来的问题是，当获取最小元素时，性能会变得很差。这是因为我们需要检查森林中的所有树，而树的总数不只是 $O(\lg n)$。

为了在常数时间获得堆顶元素，我们需要记录哪一棵树的根节点保存了最小元素。

根据这一思路，我们可以在二项式堆的基础上给出斐波那契堆定义。如下面的 Haskell 例子程序所示，我们复用二项式树的定义：

```
data BiTree a = Node { rank :: Int
                     , root :: a
                     , children :: [BiTree a]}
```

斐波那契堆要么为空，要么是一个二项式树的森林，其中含有最小元素的树被单独保存。

```
data FibHeap a = E | FH { size :: Int
                        , minTree :: BiTree a
                        , trees :: [BiTree a]}
```

方便起见，我们将堆中元素的个数也记录下来。

斐波那契堆也可以用命令式的方式定义，如下面的 C 语言例子代码。

```
struct node {
    Key key;
    struct node *next, *prev, *parent, *children;
    int degree; //即Rank

    int mark;
};

struct FibHeap {
    struct node *roots;
    struct node *minTr;
    int n; //节点的个数
};
```

上面的代码中，Key 可以是任何可比较大小的类型，简单起见我们假设类型为整数。

```
typedef int Key;
```

我们在命令式实现中，使用循环双向链表 [4]。这样可以简化很多操作并提供快速的性能。我们增加了两个额外的字段。一个是 `degree`（即 Rank），定义为一个节点中子树的数目；标志 $mark$ 仅用于减小元素值的操作。我们稍后会对 $mark$ 的作用加以介绍。

10.3.2 基本堆操作

由于斐波那契堆本质上是惰性的二项式堆，我们将复用很多二项式堆的算法。

插入新元素

可以认为二项式堆的插入算法是一种特殊的合并操作，其中一个堆仅含有一棵一个叶节点的树。

$$insert(H, x) = merge(H, singleton(x)) \tag{10.9}$$

其中 $singleton$ 是一个辅助函数，它构建出仅含有一个元素的树。

$$singleton(x) = FibHeap(1, node(1, x, \phi), \phi)$$

函数 $FibHeap()$ 接受 3 个参数，一个是大小（size），因为只有一个元素，所以值为 1，一棵特殊的树，树根存有堆中的最小元素，以及森林中剩余二项式树的列表。函数 $node()$ 和以前的含意一样，它从一个 Rank 值，一个元素，和一组子树列表构建一棵二项式树。

插入操作也可以实现为向森林中追加一个新节点，然后更新存有最小元素的树。

1: **function** INSERT(H, k)
2: $x \leftarrow$ SINGLETON(k) ▷ 将 x 装入一节点
3: append x to root list of H
4: **if** $T_{min}(H) = NIL \vee k <$ KEY($T_{min}(H)$) **then**
5: $T_{min}(H) \leftarrow x$
6: N(H) $\leftarrow$ N(H)+1

其中函数 $T_{min}()$ 返回存有最小元素的树。

下面的 C 语言例子代码实现了这一插入算法。

```
struct FibHeap* insert_node(struct FibHeap* h, struct node* x) {
    h = add_tree(h, x);
    if(h→minTr == NULL || x→key < h→minTr→key)
        h→minTr = x;
    h→n++;
    return h;
}
```

练习 10.5

选择一门命令式编程语言，实现完整的插入算法。这也是循环双向链表操作的一道习题。

堆合并

和二项式堆不同，我们在合并时并不立即进行链接操作，而是推迟到以后。这样仅仅将两个堆中的树放到一起，然后选出新的含有最小元素的树记录下来。

$$merge(H_1, H_2) = \begin{cases} H_1 & : & H_2 = \phi \\ H_2 & : & H_1 = \phi \\ FibHeap(s_1 + s_2, T_{1min}, \{T_{2min}\} \cup \mathbb{T}_1 \cup \mathbb{T}_2) & : & root(T_{1min}) < root(T_{2min}) \\ FibHeap(s_1 + s_2, T_{2min}, \{T_{1min}\} \cup \mathbb{T}_1 \cup \mathbb{T}_2) & : & otherwise \end{cases} \tag{10.10}$$

其中 s_1 和 s_2 分别时两个堆 H_1 和 H_2 的大小；T_{1min} 和 T_{2min} 分别是两个堆中存有最小的元素的树。$\mathbb{T}_1 = \{T_{11}, T_{12}, ...\}$ 是堆 H_1 的森林中其余的树；而 $\mathbb{T}_2$ 的含意类似，它包含堆 H_2 中森林里剩余的树。函数 $root(T)$ 返回一棵二项式树的根节点元素。

只要 $\cup$ 操作的性能为常数时间，合并算法的性能就是常数时间。下面的 Haskell 例子程序实现了合并操作。

```
merge h E = h
merge E h = h
merge h1@(FH sz1 minTr1 ts1) h2@(FH sz2 minTr2 ts2)
    | root minTr1 < root minTr2 = FH (sz1+sz2) minTr1 (minTr2:ts2++ts1)
    | otherwise = FH (sz1+sz2) minTr2 (minTr1:ts1++ts2)
```

命令式的合并操作可以实现为将两个堆的树连接成一个更大的列表。

```
1: function MERGE(H_1, H_2)
2:     H ← Φ
3:     ROOT(H) ← CONCAT(ROOT(H_1), ROOT(H_2))
4:     if KEY(T_min(H_1)) < KEY(T_min(H_2)) then
5:         T_min(H) ← T_min(H_1)
6:     else
7:         T_min(H) ← T_min(H_2)
       N(H) = N(H_1) + N(H_2)
8:     return H
```

这一函数假设 H_1 和 H_2 都不空。处理堆为空的情况也很容易加入，如下面的 C 例子代码所示：

```
struct FibHeap* merge(struct FibHeap* h1, struct FibHeap* h2) {
    struct FibHeap* h;
    if(is_empty(h1))
        return h2;
    if(is_empty(h2))
        return h1;
    h = empty();
    h→roots = concat(h1→roots, h2→roots);
    if(h1→minTr→key < h2→minTr→key)
        h→minTr = h1→minTr;
    else
        h→minTr = h2→minTr;
    h→n = h1→n + h2→n;
    free(h1);
    free(h2);
    return h;
}
```

使用 $merge$ 函数，可以实现常数时间 $O(1)$ 的插入算法。下面给出了常数时间 $O(1)$ 的获取顶部元素的操作。

$$top(H) = root(T_{min}) \tag{10.11}$$

练习 10.6

选择一门命令式语言，实现循环双向链表的连接。

弹出（删除最小元素）

弹出操作是斐波那契堆中最复杂的。由于在合并操作中推迟了树的链接，我们需要在其他地方将其“补偿”回来。插入，合并和 top 都已经定义好了，弹出操作是唯一剩下可以进行“补偿”的地方。

通过使用辅助数组，存在一个特别简洁的树归并（tree consolidatitoin）算法 [4]。我们稍后在命令式的实现中会介绍它。

为了实现纯函数式的树归并算法，我们先考虑这样一道关于数字的题目：

给定若干数字，例如 $\{2, 1, 1, 4, 8, 1, 1, 2, 4\}$，我们希望不断将值相同的两个数字相加，直到没有任何相等的数。这个例子的最终结果为 $\{8, 16\}$。

这个问题的解法为：

$$consolidate(L) = fold(meld, \phi, L) \tag{10.12}$$

其中 $fold()$ 遍历列表的所有元素，逐一针对每个元素和中间结果应用一个函数。它也称为reducing 操作。读者可以参考本书的附录 A，以及二叉搜索树一章。

$L = \{x_1, x_2, ..., x_n\}$ 代表要处理的数字；记 $L' = \{x_2, x_3, ..., x_n\}$ 代表除第一个数字以外的剩余数字。函数 $meld()$ 可定义如下：

$$meld(L, x) = \begin{cases} \{x\} & : & L = \phi \\ meld(L', x + x_1) & : & x = x_1 \\ \{x\} \cup L & : & x < x_1 \\ \{x_1\} \cup meld(L', x) & : & otherwise \end{cases} \tag{10.13}$$

$consolidate()$ 函数维护一个有序的结果列表 L，列表中仅包含不同的数字。L 初始化为空 ϕ。算法逐一处理每个元素 x。它首先检查 L 中的第一个元素是否等于 x，如果相等就加到一起（结果为 $2x$），然后接着判断 $2x$ 是否和 L 中的下一个元素相等。不断重复这一过程直到相加后的元素和表中第一个元素不等，或者表变为空。表10.2描述了归并序列 $\{2, 1, 1, 4, 8, 1, 1, 2, 4\}$ 的步骤。第一列表示每次“扫描”的数字；第二列是中间结果。被扫描的数字和结果列表中的第一个元素相比较。如果相等，就用两个括号围起来；最后一列是归并的结果，每个结果都用于下一步的处理。

下面的 Haskell 例子程序首先了这一数字归并的过程。

```
consolidate = foldl meld [] where
    meld [] x = [x]
    meld (x':xs) x | x == x' = meld xs (x+x')
                   | x < x'  = x:x':xs
                   | otherwise = x': meld xs x
```

我们稍后会分析归并过程的性能。

树的归并过程非常类似，唯一的区别是使用 Rank。我们需要略微修改 $meld()$ 函

数字	中间结果	结果
2	2	2
1	1, 2	1, 2
1	(1+1), 2	4
4	(4+4)	8
8	(8+8)	16
1	1, 16	1, 16
1	(1+1), 16	2, 16
2	(2+2), 16	4, 16
4	(4+4), 16	8, 16

表 10.2: 归并数字的步骤

数，使得它对 Rank 进行比较并将 Rank 相同的树链接起来。

$$meld(L, x) = \begin{cases} \{x\} & : & L = \phi \\ meld(L', link(x, x_1)) & : & rank(x) = rank(x_1) \\ \{x\} \cup L & : & rank(x) < rank(x_1) \\ \{x_1\} \cup meld(L', x) & : & otherwise \end{cases} \tag{10.14}$$

最终的树归并 Haskell 例子程序如下：

```
consolidate = foldl meld [] where
    meld [] t = [t]
    meld (t':ts) t | rank t == rank t' = meld ts (link t t')
                   | rank t <  rank t' = t:t':ts
                   | otherwise = t' : meld ts t
```

图10.9给出了斐波那契堆中树归并过程的各个步骤。和表10.2相比可以看出他们之间的相似性。

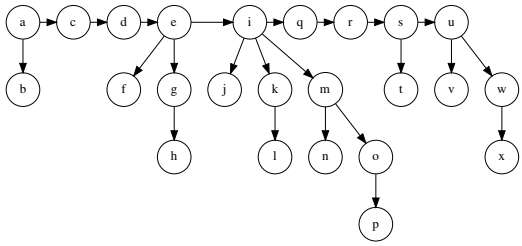
将斐波那契堆中的所有二项式树，包括存有最小元素的特殊树归并后，结果变成了二项式堆。同时我们失去了记录有最小元素的特殊树。这样就没法在常数时间 $O(1)$ 内获得堆顶元素了。

为此，需要再执行一轮 $O(\lg n)$ 的搜索，重新找到存有最小元素的树。我们可以复用前面定义的 $extractMin()$ 函数。

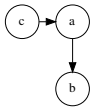
使用上面定义好的各个辅助函数，可以给出弹出操作的最终实现。记 T_{min} 为存有堆中最小元素的特殊树；$\mathbb{T}$ 为森林中除特殊树以外的剩余树，s 代表堆的大小，函数 $children()$ 返回一棵二项式树中除根以外的所有子树。

$$deleteMin(H) = \begin{cases} \phi & : & \mathbb{T} = \phi \wedge children(T_{min}) = \phi \\ FibHeap(s-1, T'_{min}, \mathbb{T}') & : & otherwise \end{cases} \tag{10.15}$$

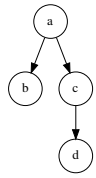
其中



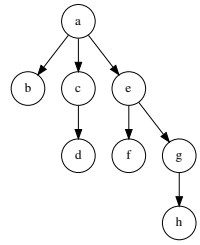
归并前



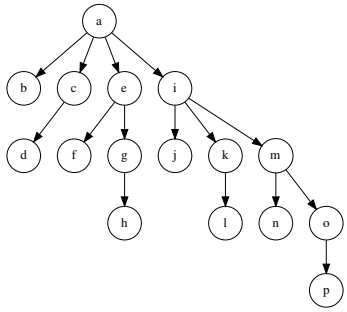
第 1、2 步



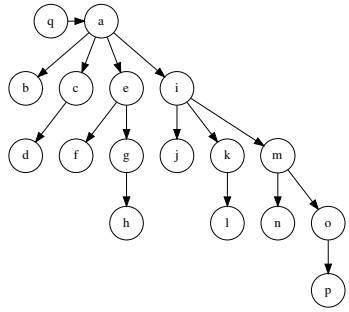
第 3 步，‘d’ 先链接到 ‘c’，然后链接到 ‘a’。



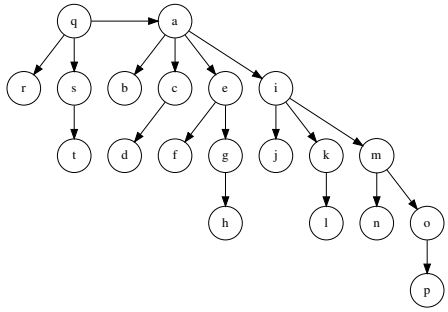
第 4 步



第 5 步



第 6 步



第 7、8 步，‘r’ 先链接到 ‘q’，然后 ‘s’ 链接到 ‘q’。

图 10.9: 树归并的步骤

$$(T'_{min}, \mathbb{T}') = extractMin(consolidate(children(T_{min}) \cup \mathbb{T}))$$

下面的 Haskell 例子程序实现了弹出操作。

```
deleteMin (FH _ (Node _ x []) []) = E
deleteMin h@(FH sz minTr ts) = FH (sz-1) minTr' ts' where
    (minTr', ts') = extractMin $ consolidate (children minTr ++ ts)
```

命令式实现的主要部分是类似的。我们将 T_{min} 的所有子树切下来，添加到森林中，然后执行树的归并操作将 Rank 相同的树链接到一起，直到所有树的 Rank 都不同。

```
1:  function DELETE-MIN(H)
2:      x ← T_min(H)
3:      if x ≠ NIL then
4:          for each y ∈ CHILDREN(x) do
5:              append y to root list of H
6:              PARENT(y) ← NIL
7:          remove x from root list of H
8:          N(H) ← N(H) - 1
9:          CONSOLIDATE(H)
10:     return x
```

算法 CONSOLIDATE 使用一个辅助数组 A 来进行归并。$A[i]$ 被定义为保存 Rank（degree）为 i 的树。在遍历森林中的树时，如果发现另外一棵 Rank 为 i 的树，我们就将它们链接起来得到一棵 Rank 为 $i+1$ 的树。然后将 $A[i]$ 清除，并接着检查 $A[i+1]$ 是否为空，若不为空，就进行后继的链接。当遍历完森林中的树后，数组 A 中就保存有最终归并后的结果。我们可以从 A 构造出斐波那契堆。

```
1:  function CONSOLIDATE(H)
2:      D ← MAX-DEGREE(N(H))
3:      for i ← 0 to D do
4:          A[i] ← NIL
5:      for each x ∈ root list of H do
6:          remove x from root list of H
7:          d ← DEGREE(x)
8:          while A[d] ≠ NIL do
9:              y ← A[d]
10:             x ← LINK(x, y)
11:             A[d] ← NIL
12:             d ← d + 1
13:         A[d] ← x
14:     T_min(H) ← NIL                    ▷ 此时 root 列表为空（NIL）
```

15: **for** $i \leftarrow 0$ to D **do**
16: **if** $A[i] \neq NIL$ **then**
17: append $A[i]$ to root list of H.
18: **if** $T_{min} = NIL \vee$ KEY$(A[i]) <$ KEY$(T_{min}(H))$ **then**
19: $T_{min}(H) \leftarrow A[i]$

这里唯一没有确定的算法是 MAX-DEGREE，它可以确定斐波那契堆中任何节点的 degree 上限。我们将在最后一节中给出它的实现。

用上述算法处理图??中所示的斐波那契堆，各个步骤中的数组 A 如图10.10所示。

下面的 C 语言例子程序实现了上述算法。

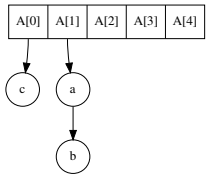
```
void consolidate(struct FibHeap* h) {
    if(!h→roots)
        return;
    int D = max_degree(h→n)+1;
    struct node *x, *y;
    struct node** a = (struct node**)malloc(sizeof(struct node*)*(D+1));
    int i, d;
    for(i=0; i≤D; ++i)
        a[i] = NULL;
    while(h→roots) {
        x = h→roots;
        h→roots = remove_node(h→roots, x);
        d= x→degree;
        while(a[d]) {
            y = a[d];  //存在和x的degree相等的另一节点。
            x = link(x, y);
            a[d++] = NULL;
        }
        a[d] = x;
    }
    h→minTr = h→roots = NULL;
    for(i=0; i≤D; ++i)
        if(a[i]) {
            h→roots = append(h→roots, a[i]);
            if(h→minTr == NULL || a[i]→key < h→minTr→key)
                h→minTr = a[i];
        }
    free(a);
}
```

练习 10.7

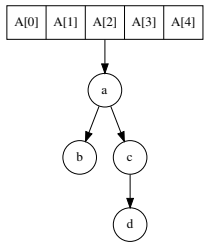
选择一门命令式编程语言，实现循环双向链表的节点删除程序。

10.3.3 弹出操作的性能分析

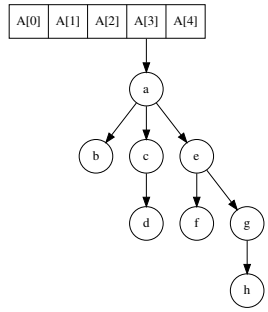
为了分析弹出算法的分摊性能，需要使用“势方法（potential method）”。读者可以参考 [4] 了解这一方法的严格定义。这里我们仅仅给出一个直观的描述。



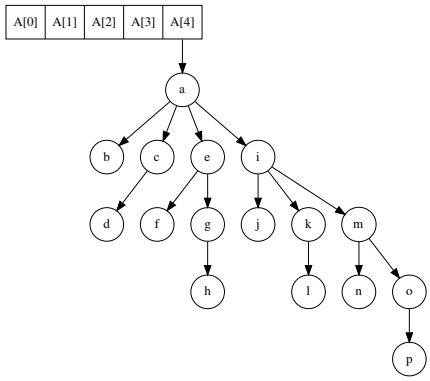
第 1、2 步



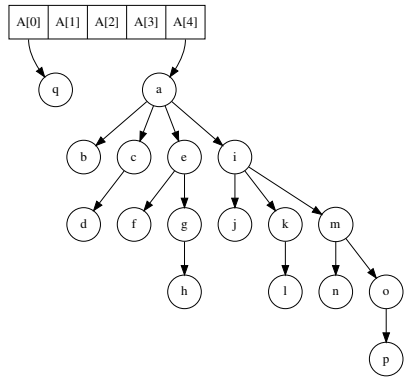
第 3 步，因为 $A_0 \neq NIL$，‘d’ 先被链接到 ‘c’，然后 A_0 被清除为 NIL。接下来，由于 $A_1 \neq NIL$，‘c’ 被链接到 ‘a’，新的树被存入 A_2。



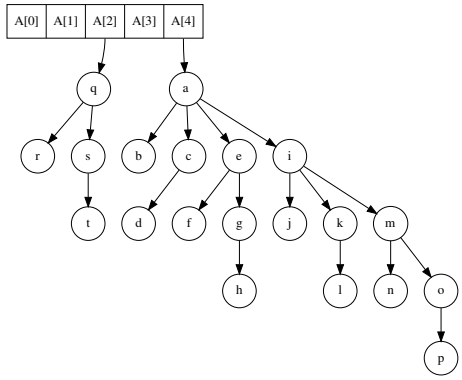
第 4 步



第 5 步



第 6 步



第 7、8 步，因为 $A_0 \neq NIL$，‘r’ 被链接到 ‘q’，新树存入 A_1（A_0 清空）；接着 ‘s’ 链接到 ‘q’，并存入 A_2（A_1 清空）。

图 10.10: 树归并的步骤

回忆物理学中关于重力势能的定义：

$$E = M \cdot g \cdot h$$

假设一个复杂的操作过程，将质量为 M 的物体上下移动，最终物体静止在了高为 h' 的位置。如果这一过程中的摩擦阻力做功 W_f，则做功的总和为：

$$W = M \cdot g \cdot (h' - h) + W_f$$

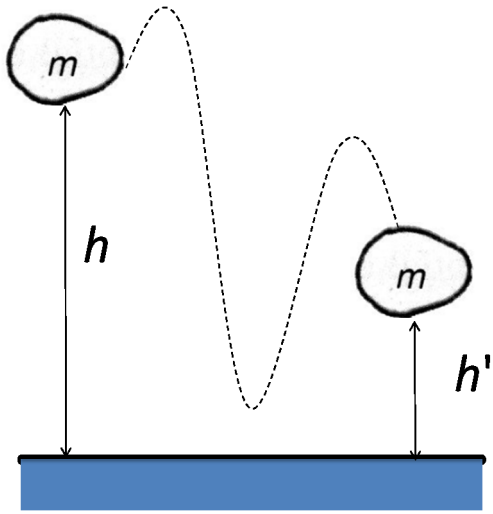


图 10.11: 重力势能

图10.11描述了这一概念。

我们用同样的方法来考虑斐波那契堆的弹出操作，为了计算总消耗，我们首先定义删除最小元素前的势为 $\Phi(H)$。这个势是由迄今为止的插入和合并操作累积的。经过树的归并操作，我们得到了新的堆 H'，由此计算新的势 $\Phi(H')$。两个势 $\Phi(H')$ 和 $\Phi(H)$ 的差再加上树归并算法消耗的部分就可以给出弹出操作的分摊复杂度。

为了分析弹出操作，定义势为：

$$\Phi(H) = t(H) \tag{10.16}$$

其中 $t(H)$ 是斐波那契堆森林中树的棵数。对于任何非空的堆，我们有 $t(H) = 1 + length(\mathbb{T})$。

对于 n 个节点的斐波那契堆，设所有树的 Rank 上限为 $D(n)$。经过归并，保证堆森林中树的棵数最多为 $D(n) + 1$。

在归并前，我们还做了另外一个重要的操作，也对总运行时间有所贡献：我们将存有最小元素的树根删除，然后将其全部子树添加到森林中。因此树归并操作最多处理 $D(n) + t(H) - 1$ 棵树。

总结上述各个因素，我们可以推导出分摊性能如下：

$$\begin{aligned} T &= T_{consolidation} + \Phi(H') - \Phi(H) \\ &= O(D(n) + t(H) - 1) + (D(n) + 1) - t(H) \\ &= O(D(n)) \end{aligned} \tag{10.17}$$

如果只执行过插入、合并和弹出操作，可以确保斐波那契堆中的所有树都为二项式树。因此可以很容易地估计出 $D(n)$ 的上限为 $O(\lg n)$（考虑极端情况，所有的节点都在唯一的一棵二项式树中）。

但是，接下来的一节中我们会介绍，存在一种操作会破坏树为二项式树的约定。

练习 10.8

为何树归并操作的时间和它处理的树的数目成比例？

10.3.4 减小 key

还有一种特殊的堆操作，它只在命令式的环境下存在。这个操作就是将某个节点的值减小。减小节点的值对于某些图算法，例如最小生成树算法和 Dijkstra 算法非常重要 [4]，而且我们需要这一操作的分摊性能达到常数时间 $O(1)$。

但是我们无法定义一个高效的函数 $Decrease(H, k, k')$，使得它先定位到 key 等于 k 的节点，然后将 k 替换为 k'，最后再恢复堆性质。这是由于如果没有指向目标节点的引用，定位一个节点的时间为 $O(n)$。

在命令式的环境中，我们可以定义算法 DECREASE-KEY(H, x, k)。其中 x 是堆 H 中一个节点的引用，我们希望将它的值减小到 k。使用 x，我们无需再执行查找操作，因此有可能给出分摊复杂度为常数时间 $O(1)$ 的算法。

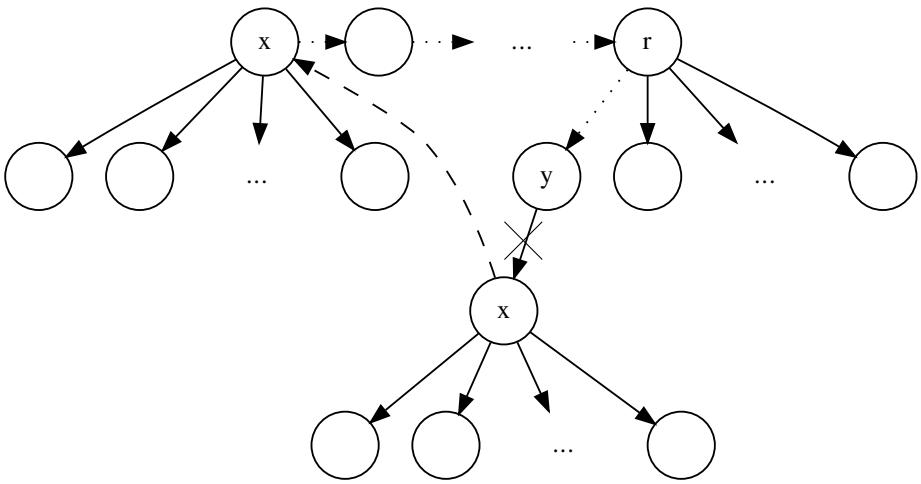


图 10.12: $x < y$，将子树 x 从其父节点上切下，然后添加到森林中。

图10.12描述了这一情况。将节点 x 的值减小后，它小于 y，我们将 x 从其父节点 y 切下，然后将根为 x 的树“粘贴”到森林中。

虽然我们恢复了堆性质使得父节点的值小于所有的子树，但是由于切除了某些子树，它不再是一棵二项式树了。如果一棵树被切除了很多子树，就无法保证合并操作的性能了。为了避免这一问题，斐波那契堆增加了另外一个限制条件：

“如果一个节点失去了它的第二个子节点，它被立即从父节点切下，然后添加到森林中。”

最终的 DECREASE-KEY 算法实现如下：

```
1: function DECREASE-KEY(H, x, k)
2:     KEY(x) ← k
3:     p ← PARENT(x)
4:     if p ≠ NIL ∧ k < KEY(p) then
5:         CUT(H, x)
6:         CASCADING-CUT(H, p)
7:     if k < KEY(T_min(H)) then
8:         T_min(H) ← x
```

其中函数 CASCADING-CUT 使用一个标记来记录它是否失去第二个子节点。当节点失去第一个子节点时被加上这个标记。在函数 CUT 中清除这一标记。

```
1: function CUT(H, x)
2:     p ← PARENT(x)
3:     remove x from p
4:     DEGREE(p) ← DEGREE(p) - 1
5:     add x to root list of H
6:     PARENT(x) ← NIL
7:     MARK(x) ← FALSE
```

在级联切除（cascading cut）过程中，若节点 x 被标记了，说明它已经失去了一个子节点。我们递归对其父节点执行切除和级联切除直到达到根节点。

```
1: function CASCADING-CUT(H, x)
2:     p ← PARENT(x)
3:     if p ≠ NIL then
4:         if MARK(x) = FALSE then
5:             MARK(x) ← TRUE
6:         else
7:             CUT(H, x)
8:             CASCADING-CUT(H, p)
```

下面的 C 语言例子程序实现了减小 key 的算法。

```
void decrease_key(struct FibHeap* h, struct node* x, Key k) {
```

```
    struct node* p = x->parent;
    x->key = k;
    if(p && k < p->key) {
        cut(h, x);
        cascading_cut(h, p);
    }
    if(k < h->minTr->key)
        h->minTr = x;
}

void cut(struct FibHeap* h, struct node* x) {
    struct node* p = x->parent;
    p->children = remove_node(p->children, x);
    p->degree--;
    h->roots = append(h->roots, x);
    x->parent = NULL;
    x->mark = 0;
}

void cascading_cut(struct FibHeap* h, struct node* x) {
    struct node* p = x->parent;
    if(p) {
        if(!x->mark)
            x->mark = 1;
        else {
            cut(h, x);
            cascading_cut(h, p);
        }
    }
}
```

练习 10.9

证明 Decrease-Key 算法的分摊复杂度为常数时间 $O(1)$。

10.3.5 斐波那契堆名字的由来

最后，我们来解释为什么这个数据结构的名字叫作“斐波那契堆”。

我们还剩下一个算法没有给出实现 Max-Degree(n)。它用来给出含有 n 个节点的斐波那契堆中任意节点 degree 的上限。我们将用斐波那契数列的性质来给出证明，并最终实现 Max-Degree 算法。

引理 10.3.1. 堆斐波那契堆中的任何节点 x，记 $k = degree(x)$，$|x| = size(x)$，存在以下关系：

$$|x| \geq F_{k+2} \tag{10.18}$$

其中 F_k 为斐波那契数列：

$$F_k = \begin{cases} 0 & : & k = 0 \\ 1 & : & k = 1 \\ F_{k-1} + F_{k-2} & : & k \geq 2 \end{cases}$$

证明. 考虑节点 x 的全部 k 棵子树，将它们记为：$y_1, y_2, ..., y_k$，顺序按照它们被链接到 x 时间的先后。其中 y_1 是最早被加入的，而 y_k 是最新加入的。

显然有 $|y_i| \geq 0$。当 y_i 链接到 x 的时候，子树 $y_1, y_2, ..., y_{i-1}$ 已经存在了。因为算法只会把 Rank 相同的树链接起来，所以在这一时刻，我们有：

$$degree(y_i) = degree(x) = i - 1$$

此后, 节点 y_i 最多只能失去一个子节点 (通过减小 key 操作), 否则一旦失去第二个子节点, 它会被立即切除并加入到森林中。因此我们可以推断, 对任何 $i = 2, 3, ..., k$, 有：

$$degree(y_i) \geq i - 2$$

令 s_k 为节点 x 含有子节点个数可能的最小值，其中 $degree(x) = k$。对于边界情况，有 $s_0 = 1$, $s_1 = 2$，对于其他情况，可以推出：

$$\begin{aligned} |x| &\geq s_k \\ &= 2 + \sum_{i=2}^{k} s_{degree(y_i)} \\ &\geq 2 + \sum_{i=2}^{k} s_{i-2} \end{aligned}$$

我们接下来要证明 $s_k > F_{k+2}$。使用数学归纳法。对于边界情况，我们有 $s_0 = 1 \geq F_2 = 1$，以及 $s_1 = 2 \geq F_3 = 2$。对于 $k \geq 2$ 的情况，我们有：

$$\begin{aligned} |x| &\geq s_k \\ &\geq 2 + \sum_{i=2}^{k} s_{i-2} \\ &\geq 2 + \sum_{i=2}^{k} F_i \\ &= 1 + \sum_{i=0}^{k} F_i \end{aligned}$$

现在，我们需要证明

$$F_{k+2} = 1 + \sum_{i=0}^{k} F_i \tag{10.19}$$

再次使用数学归纳法：

- 边界情况：$F_2 = 1 + F_0 = 2$
- 递归情况：

$$\begin{aligned} F_{k+2} &= F_{k+1} + F_k \\ &= 1 + \sum_{i=0}^{k-1} F_i + F_k \\ &= 1 + \sum_{i=0}^{k} F_i \end{aligned}$$

综上，我们得到最终结论：

$$n \geq |x| \geq F_k + 2 \tag{10.20}$$

□

回忆 AVL 树的结果：$F_k \geq \phi^k$，其中 $\phi = \frac{1+\sqrt{5}}{2}$ 为黄金分割比。我们同时证明了弹出操作的分摊复杂度为 $O(\lg n)$。

根据这一结果，我们可以定义函数 $MaxDegree$ 如下：

$$MaxDegree(n) = 1 + \lfloor \log_\phi n \rfloor \tag{10.21}$$

命令式的 Max-Degree 算法可以同样使用斐波那契数列来实现：

```
1: function MAX-DEGREE(n)
2:     F_0 ← 0
3:     F_1 ← 1
4:     k ← 2
5:     repeat
6:         F_k ← F_{k_1} + F_{k_2}
7:         k ← k + 1
8:     until F_k < n
9:     return k − 2
```

下面的 C 例子程序实现了这一算法：

```
int max_degree(int n) {
    int k, F;
    int F2 = 0;
    int F1 = 1;
    for(F = F1 + F2, k = 2; F < n; ++k) {
        F2 = F1;
        F1 = F;
        F = F1 + F2;
    }
    return k-2;
}
```

10.4 配对堆

虽然斐波那契堆在理论上有着优异的性能，但是它的实现复杂。人们发现斐波那契堆复杂度中 big-O 后面的常数较大，它的理论意义要大于实际意义。

本节中，我们介绍另外一种堆 配对（pairing）堆。它是已知性能最好的堆。大部分操作，包括插入、获取顶部元素、合并都是常数时间 $O(1)$ 的，人们猜测它的弹出操作的分摊复杂度为 $O(\lg n)$[58][3]。到作者书写本章为止的 15 年内，这一猜想还没有得到证明。尽管有大量的试验数据支持它的分摊复杂度为 $O(\lg n)$。

除了性能优异，配对堆还很简单。存在简洁的命令式和函数式实现。

10.4.1 定义

二项式堆和斐波那契堆都由森林来实现。而配对堆本质上是一棵 K 叉树。最小元素保存于树根，其余元素存储于子树中。

下面的 Haskell 程序定义了配对堆。

```
data PHeap a = E | Node a [PHeap a]
```

这是一个递归定义，一个配对堆要么为空，要么是一棵 K 叉树，包含一个根节点和一组子树。

下面的 C 语言例子程序，也给出了配对堆的定义。简单起见，我们仅仅讨论最小堆，并且假设 key 的类型为整数[2]。我们使用单向链表表示的“左侧孩子，右侧兄弟”定义（二叉树表示法 [4]）。

```
typedef int Key;

struct node {
    Key key;
    struct node *next, *children, *parent;
};
```

其中的父节点字段仅在减小 key 值的操作中用到，其他情况下可以忽略。我们稍后会加以解释。

10.4.2 基本堆操作

我们首先介绍堆的合并操作，合并操作可以用以实现插入。获取顶部元素相对简单。而弹出操作则较为复杂。

合并、插入、和获取顶部元素

合并操作的思想和二项式堆的链接相似。当我们合并两个配对堆时，存在两种情况：

[2]可以将 key 的类型抽象为 C++ 的模板参数，读者可以参考本书附带的例子程序。

- 简单情况：其中一个堆为空，我们只要返回另一堆作为合并的结果；
- 否则，我们比较两个堆的根节点元素，令根节点较大的一个作为另一个的新子树。

令 H_1 和 H_2 分别代表两个堆，x 和 y 为各自的根节点元素。函数 $Children()$ 返回一棵 K 叉树的子树，$Node()$ 从一个根节点元素和一组子树构造一棵 K 叉树。

$$merge(H_1, H_2) = \begin{cases} H_1 & : & H_2 = \phi \\ H_2 & : & H_1 = \phi \\ Node(x, \{H_2\} \cup Children(H_1)) & : & x < y \\ Node(y, \{H_1\} \cup Children(H_2)) & : & otherwise \end{cases} \tag{10.22}$$

其中

$$x = Root(H_1)$$
$$y = Root(H_2)$$

显然合并算法的性能为常数时间 $O(1)$[3]。

下面的 Haskell 例子程序给出了 $merge$ 操作。

```
merge h E = h
merge E h = h
merge h1@(Node x hs1) h2@(Node y hs2) =
    if x < y then Node x (h2:hs1) else Node y (h1:hs2)
```

也可以用命令式的方式实现合并操作。使用“左侧孩子，右侧兄弟”方法，我们可以把根节点元素较大的堆，利用链表操作链接到另一个堆的子树前面变成第一棵子树。这一常数时间的操作可以描述如下：

```
1: function MERGE(H1, H2)
2:     if H1 = NIL then
3:         return H2
4:     if H2 = NIL then
5:         return H1
6:     if KEY(H2) < KEY(H1) then
7:         EXCHANGE(H1 ↔ H2)
8:     Insert H2 in front of CHILDREN(H1)
9:     PARENT(H2) ← H1
10:    return H1
```

在这一过程中，我们也更新了指向父节点的字段。相应的 C 语言例子代码如下：

```
struct node* merge(struct node* h1, struct node* h2) {
    if (h1 == NULL)
```

[3]假设 $\cup$ 操作的性能为常数时间。对于单向链表的“cons”来说，这一点成立。

```
        return h2;
    if (h2 == NULL)
        return h1;
    if (h2→key < h1→key)
        swap(&h1, &h2);
    h2→next = h1→children;
    h1→children = h2;
    h2→parent = h1;
    h1→next = NULL; /*Break previous link if any*/
    return h1;
}
```

其中 swap 函数和斐波那契堆中的定义类似。

使用合并函数，可以像斐波那契堆的式 (10.9) 一样实现插入操作。这一操作的性能为常数时间 $O(1)$。因为最小的元素总是保存在根节点，所以可以通过根来获取顶部元素。

$$top(H) = Root(H) \tag{10.23}$$

获取顶部元素的性能也是常数时间 $O(1)$ 的。

减小节点的值

同斐波那契堆一样，减小节点的值仅在命令式环境下有意义。这一问题的解比斐波那契堆要简单。节点的值减小后，我们将它为根的子树从父节点上切下，然后合并到堆中。唯一特殊的情况是，如果是根节点，我们可以直接改变值而无需做任何额外操作。

下面的算法描述了将节点 x 的值减小到 k 的操作。

1: **function** DECREASE-KEY(H, x, k)
2: $\quad$ KEY(x) $\leftarrow k$
3: $\quad$ **if** PARENT(x) $\neq$ NIL **then**
4: $\quad\quad$ Remove x from CHILDREN(PARENT(x)) PARENT(x) $\leftarrow$ NIL
5: $\quad\quad$ **return** MERGE(H, x)
6: $\quad$ **return** H

下面的 C 语言例子程序实现了这一算法。

```
struct node* decrease_key(struct node* h, struct node* x, Key key) {
    x→key = key; /* Assume key ≤ x→key */
    if(x→parent) {
        x→parent→children = remove_node(x→parent→children, x);
        x→parent = NULL;
        return merge(h, x);
    }
    return h;
}
```

练习 10.10

选择一门语言，实现从父节点将子树切下的操作。考虑如何保证减小 key 操作的总体性能为常数时间 $O(1)$？仅仅使用“左侧孩子，右侧兄弟”就够了么？

弹出

因为最小元素总是保存在根节点，弹出操作将其删除后，会剩下一系列子树。这些子树可以合并成一棵较大的树。

$$pop(H) = mergePairs(Children(H)) \tag{10.24}$$

配对堆使用一种特殊的合并策略，它先从左向右，两两成对地将子树合并。然后从右向左将子树对合并的结果再次合并成一棵树。配对堆的名字就来自这一合并过程。

图10.13和10.14描述了这一成对合并的过程。

递归地成对合并过程和自底向上的归并排序 [3] 类似。记配对堆的全部子树为 A，它是一个子树的列表 $\{T_1, T_2, T_3, ..., T_m\}$。$mergePairs()$ 函数定义如下：

$$mergePairs(A) = \begin{cases} \Phi & : & A = \Phi \\ T_1 & : & A = \{T_1\} \\ merge(merge(T_1, T_2), mergePairs(A')) & : & otherwise \end{cases} \tag{10.25}$$

其中

$$A' = \{T_3, T_4, ..., T_m\}$$

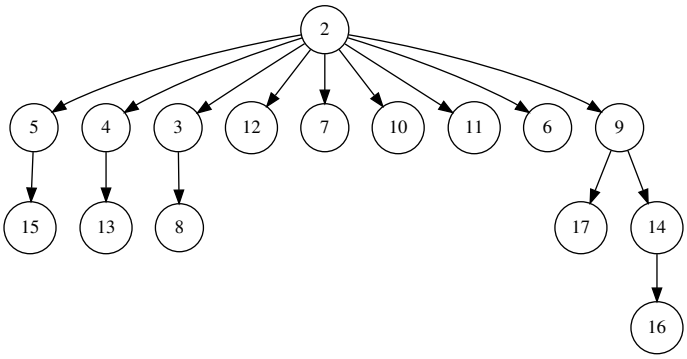
是除去前两棵树外剩余的子树。

下面的 Haskell 例子程序实现了这一算法。

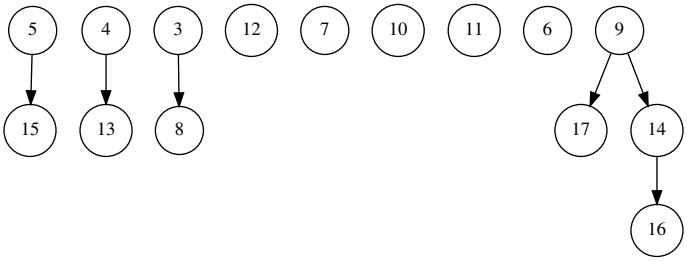
```
deleteMin (Node _ hs) = mergePairs hs where
    mergePairs [] = E
    mergePairs [h] = h
    mergePairs (h1:h2:hs) = merge (merge h1 h2) (mergePairs hs)
```

弹出操作也可以按照过程描述如下：

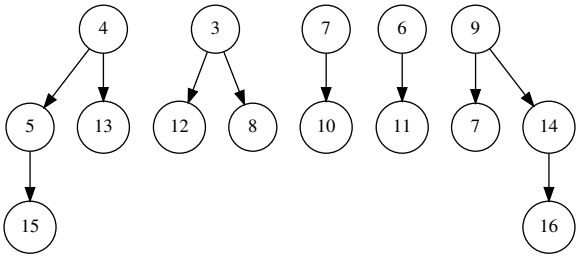
```
1: function POP(H)
2:     L ← NIL
3:     for every 2 trees T_x, T_y ∈ CHILDREN(H) from left to right do
4:         Extract x, and y from CHILDREN(H)
5:         T ← MERGE(T_x, T_y)
6:         Insert T at the beginning of L
7:     H ← CHILDREN(H)                          ▷ H is either NIL or one tree.
8:     for ∀T ∈ L from left to right do
```



(a) 弹出前的配对堆

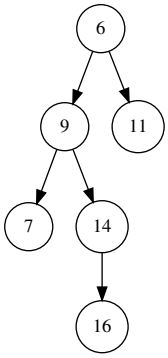


(b) 根节点 2 被删除，剩余 9 棵子树

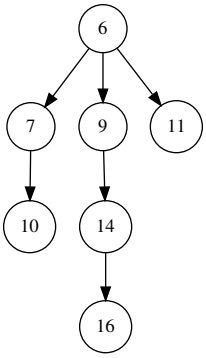


(c) 每两棵树成对合并，因为有奇数棵树，所以最后一棵无需合并。

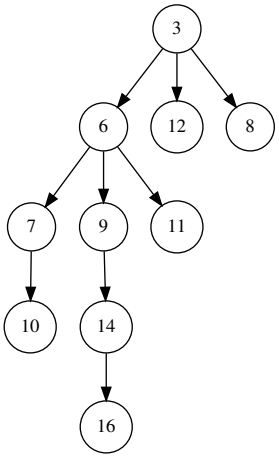
图 10.13: 删除根节点，将子树成对合并



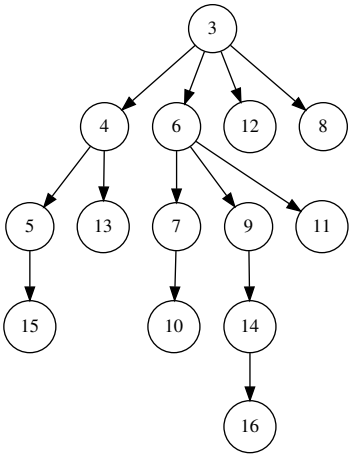
(a) 将根节点为 9 和 6 的两棵树合并



(b) 将根节点为 7 的树合并到当前结果中



(c) 将根节点为 3 的树合并到结果中



(d) 将根节点为 4 的树合并到结果中

图 10.14: 从右向左合并的步骤

9: $H \leftarrow \text{MERGE}(H, T)$

10: **return** H

其中 L 被初始化为一个空的链表，然后算法从左向右每次成对迭代 K 叉树中的两棵子树，进行合并。结果被插入到 L 的头部。因为在链表的前方插入，所以再次遍历链表 L 时，实际是按照从右向左的顺序。堆 H 中可能含有奇数棵子树，这种情况下，成对合并后会剩余一棵。处理办法是从这一棵树开始进行从右向左的合并。

下面的 C 语言例子程序实现了弹出算法。

```
struct node* pop(struct node* h) {
    struct node *x, *y, *lst = NULL;
    while ((x = h→children) ≠ NULL) {
        if ((h→children = y = x→next) ≠ NULL)
            h→children = h→children→next;
        lst = push_front(lst, merge(x, y));
    }
    x = NULL;
    while((y = lst) ≠ NULL) {
        lst = lst→next;
        x = merge(x, y);
    }
    free(h);
    return x;
}
```

人们猜想配对堆的弹出操作的分摊性能为 $O(\lg n)$[58]。

练习 10.11

选择一门语言，实现在链表的头部插入一棵树。

删除节点

我们没有在二项式堆和斐波那契堆提到删除操作。删除可以实现为先将节点的值减小为负无穷（$-\infty$），然后再执行一次弹出操作。这里我们介绍另外一种删除方法。

我们需要定义函数 $delete(H, x)$，其中 x 是配对堆 H 中的某一节点[4]。

若 x 为根节点，我们只需要执行一次弹出操作。否则，我们将 x 从 H 中切下，然后对 x 执行一次弹出操作，再将弹出结果合并回 H。如下：

$$delete(H, x) = \begin{cases} pop(H) & : \quad x \quad \text{is root of} \quad H \\ merge(cut(H, x), pop(x)) & : \quad otherwise \end{cases} \tag{10.26}$$

因为删除算法调用弹出操作，因此人们猜想它的分摊性能也是对数时间 $O(\lg n)$。

练习 10.12

[4] 具体来说，x 是某一节点的引用

- 选择一门语言，实现命令式的删除算法。
- 考虑如何完整实现纯函数式的删除算法。

10.5 小结

本章中，我们将堆的实现从二叉树扩展到了更加丰富的数据结构。二项式堆和斐波那契堆使用 K 叉树的森林作为底层数据结构，而配对堆使用一棵 K 叉树来存储数据。通过将某些费时的操作延迟进行，可以获得总体上优异的分摊性能。这一点很具有启发性。虽然斐波那契堆在理论上具有良好的性能，但是实现较为复杂，最近的一些教科书往往会跳过不讲。本章介绍的配对堆具备简单的实现，并且在实际应用中性能表现很好。

到本章为止，我们介绍了一些最基本的基于树的数据结构。还有大量和树有关的内容有待我们去了解和探索。从下一章开始，我们将介绍通用的序列数据结构，包括数组和队列。

第十一章　并不简单的队列

11.1　简介

队列是一种看起来比较简单的数据结构。它提供了 FIFO（先进先出）处理数据的机制。有多种方法可以实现队列，包括使用单向或双向链表，使用循环缓冲区（ring buffer）等。但是，在满足队列性质的条件下（尤其是性能限制条件），实现一个纯函数式队列却并不简单。

本章中，我们介绍实现队列的多种策略。队列是一种先进先出的数据结构，并且满足如下的性能限制：

- 可以在常数时间 $O(1)$ 内向末尾添加元素；
- 可以在常数时间 $O(1)$ 内从头部获取或删除元素。

这两条性质必须被满足。有时还会增加一些目标，例如能够动态分配内存。

显然，双向链表可以很容易地实现队列。但是还存在更加简单的方案，队列可以由单向链表或者普通数组实现。我们这里要提出的问题是：如何实现一个纯函数式的队列?

我们将首先解释典型的单向链表和循环缓冲区实现的队列；然后我们给出一个简单直观的函数式实现，但是这一解法的性能只在分摊的意义下能达到常数时间。我们还将介绍实时（或最坏情况下）性能达到常数时间的实现方法。最后，我们介绍一种依赖于惰性求值的实时队列。

大部分函数式实现来自 Chris Okasaki 的工作 [3]，他给出了 16 种不同的纯函数式队列。

11.2　单向列表和循环缓冲区实现的队列

本节我们分别介绍用单项链表实现的队列，和用循环缓冲区实现的队列。它们是典型的命令式队列。

11.2.1 单向链表实现

使用单向链表，可以很容易地在链表的头部以常数时间 $O(1)$ 插入或删除元素。但是为了保证先进先出，只能在链表头部执行一种操作，而在链表的尾部执行相反的另一种操作。

对于单向链表，为了在尾部执行操作，我们需要遍历整个链表以到达尾部。遍历需要 $O(n)$ 时间，其中 n 是链表长度。这样就无法达到队列的性能要求。

为了解决这个问题，我们需要一个额外的记录来快速访问链表的尾部。使用一个 sentinel 可以简化边界的处理。下面的 C 语言例子程序使用单向链表实现了队列[1]。

```
typedef int Key;

struct Node {
  Key key;
  struct Node* next;
};

struct Queue {
  struct Node *head, *tail;
};
```

图11.1描述了一个空链表。头部和尾部都指向空的 sentinel 节点。

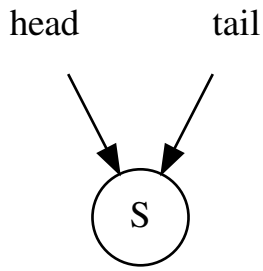


图 11.1: 空队列，头部和尾部都指向 sentinel 节点

我们将队列的接口抽象如下：

function EMPTY ▷ 创建一个空队列

function EMPTY?(Q) ▷ 检查一个队列 Q 是否为空

function ENQUEUE(Q, x) ▷ 将新元素 x 加入队列 Q（入队）

function DEQUEUE(Q) ▷ 从队列 Q 删除一个元素（出队）

function HEAD(Q) ▷ 按照先进先出顺序获取队列 Q 中的下一个元素

注意 DEQUEUE 和 HEAD 的区别。HEAD 仅仅按照 FIFO 的顺序获取下一个元素而不会将元素删除，而 DEQUEUE 会执行删除操作。

[1]使用 C++ 模板可以抽象元素的类型。简单起见，C 语言例子程序种假设元素为整数。

某些编程语言，如 Haskell 和大多数面向对象的语言，可以通过某种形式保证上述接口。下面的 Haskell 例子代码定义了抽象队列。

```
class Queue q where
    empty :: q a
    isEmpty :: q a → Bool
    push :: q a → a → q a      — 或命名为: snoc、append、push_back
    pop :: q a → q a           — 或命名为: tail、pop_front
    front :: q a → a           — 或命名为: head
```

为了保证 ENQUEUE 和 DEQUEUE 可以在常数时间内完成，我们在头部加入元素，从尾部删除元素[2]。

function ENQUEUE(Q, x)
  $p \leftarrow$ CREATE-NEW-NODE
  KEY(p) $\leftarrow x$
  NEXT(p) $\leftarrow NIL$
  NEXT(TAIL(Q)) $\leftarrow p$
  TAIL(Q) $\leftarrow p$

因为使用 sentinel 节点，所以至少存在一个节点（空队列中有一个 sentinel 节点）。因此上述算法在追加新节点 p 时，无需检查尾部是否有效。

function DEQUEUE(Q)
  $x \leftarrow$ HEAD(Q)
  NEXT(HEAD(Q)) $\leftarrow$ NEXT(x)
  if $x =$ TAIL(Q) **then** ▷ Q 变为空
    TAIL(Q) $\leftarrow$ HEAD(Q)
  return KEY(x)

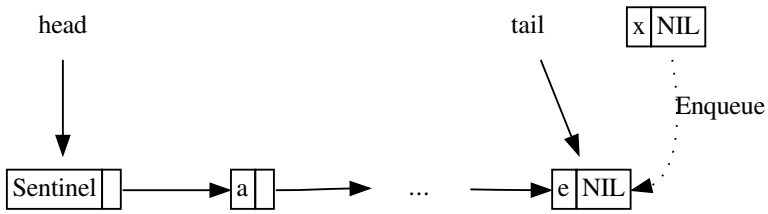
因为我们总是保证 sentinel 节点在所有其他节点的前面，函数 HEAD 实际返回 sentinel 的下一个节点。

图11.2描述了 ENQUEUE 和 DEQUEUE 使用 sentinel 节点工作的情形。

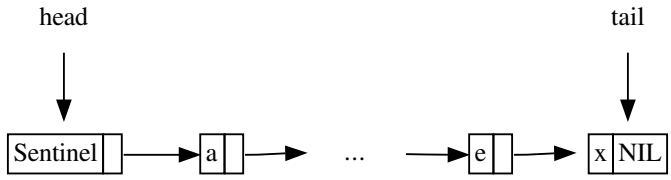
下面的 C 语言例子程序实现了入队和出队算法。

```
struct Queue* enqueue(struct Queue* q, Key x) {
    struct Node* p = (struct Node*)malloc(sizeof(struct Node));
    p→key = x;
    p→next = NULL;
    q→tail→next = p;
    q→tail = p;
    return q;
}

Key dequeue(struct Queue* q) {
    struct Node* p = head(q); //获取sentinel的下一个节点。
    Key x = key(p);
```

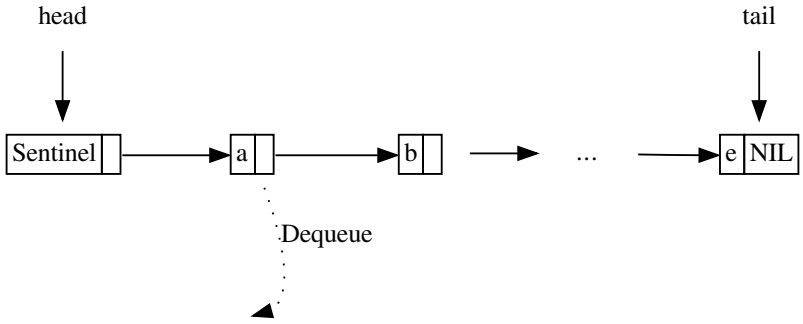
[2]也可以在尾部加入，从头部删除，但是相应的操作会变复杂。



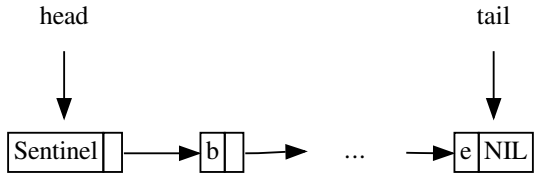
(a) 执行 Enqueue 前



(b) 执行 Enqueue 后



(c) 执行 Dequeue 前



(d) 执行 Dequeue 后

图 11.2: 单向链表实现队列的 Enqueue 和 Dequeue 操作

```
    q→head→next = p→next;
    if(q→tail == p)
        q→tail = q→head;
    free(p);
    return x;
}
```

这一方案简单、稳健。可以很容易地扩展到并发环境（例如多核）。我们可以在头部和尾部各使用一把锁。sentinel 节点可以帮助我们在队列为空时避免死锁 [59]、[60]。

练习 11.1

- 使用单向链表，实现 EMPTY? 和 HEAD 操作。
- 选择一门命令式语言，用单向链表实现队列。注意提供初始化和释放队列的函数。

11.2.2 循环缓冲区实现

另外一种方法是使用数组来实现一个循环缓冲区（也称 ring buffer）。和单向链表正相反，空间足够的情况下，数组支持在尾部进行常数时间的追加操作。如果空间不足，数组已满，我们需要重新申请空间。但是从数组头部删除元素的性能较差，为线性时间 $O(n)$。这是因为需要将剩余的全部元素向前移动（shift）一个单元以添补删除第一个元素后的空位。

循环缓冲区的思路如图11.3和11.4所示。

下面的 C 语言例子程序使用循环缓冲区实现了队列，它规定了缓冲区的最大容量，而没有使用动态分配内存。

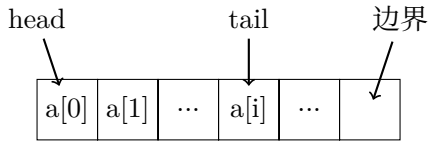
```
struct QueueBuf{
    Key* buf;
    int head, cnt, size;
};
```

在队列初始化时，传入队列的容量参数。

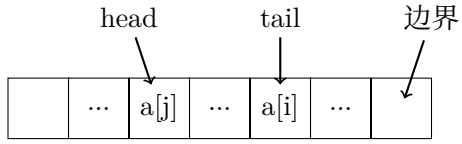
```
struct QueueBuf* createQ(int max){
    struct QueueBuf* q = (struct QueueBuf*)malloc(sizeof(struct QueueBuf));
    q→buf = (Key*)malloc(sizeof(Key)*max);
    q→size = max;
    q→head = q→cnt = 0;
    return q;
}
```

可以使用一个变量来记录队列中存储的元素个数，并用来判断队列是否为空，或者是否已满。

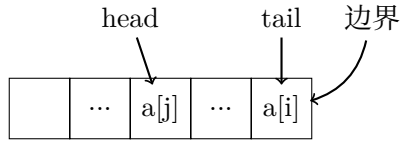
function EMPTY?(Q)



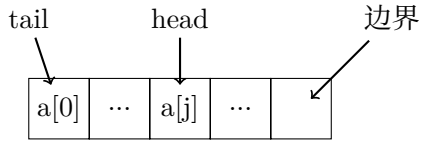
(a) 连续加入多个元素。



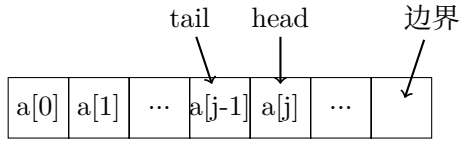
(b) 从头部删除若干元素后，出现了空档。



(c) 继续加入多个元素直到数组的边界。



(d) 下一个元素加入到数组头部的第一个单元。



(e) 全部单元都保存了元素，队列已满。

图 11.3: 使用循环缓冲区实现队列

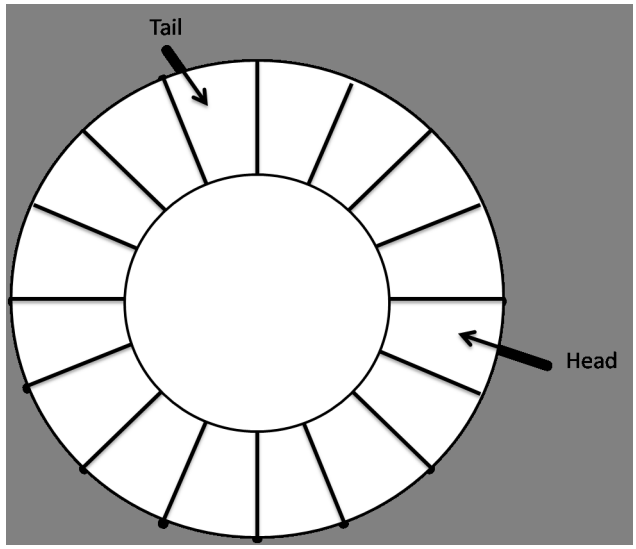


图 11.4: 循环缓冲区

return COUNT(Q) = 0

实现入队 ENQUEUE 和 DEQUEUE 出队操作的最简单方法是使用模运算。

function ENQUEUE(Q, x)
  if ¬ FULL?(Q) **then**
    COUNT(Q) ← COUNT(Q) + 1
    tail ← (HEAD(Q) + COUNT(Q)) mod SIZE(Q)
    BUFFER(Q)[tail] ← x

function HEAD(Q)
  if ¬ EMPTY?(Q) **then**
    return BUFFER(Q)[HEAD(Q)]

function DEQUEUE(Q)
  if ¬ EMPTY?(Q) **then**
    HEAD(Q) ← (HEAD(Q) + 1) mod SIZE(Q)
    COUNT(Q) ← COUNT(Q) - 1

但取模运算在某些环境下很慢，可以通过一些调整避免取模运算，如下面的 C 语言例子程序所示：

```
void enQ(struct QueueBuf* q, Key x){
    if(!fullQ(q)){
        q→buf[offset(q→head + q→cnt, q→size)] = x;
        q→cnt++;
    }
}
```

```
Key headQ(struct Queue* q) {
    return q->buf[q->head];  //假设队列不为空。
}

Key deQ(struct QueueBuf* q){
    Key x = headQ(q);
    q->head = offset(++q->head, q->size);
    q->cnt--;
    return x;
}
```

练习 11.2

循环缓冲区的队列在初始化时规定了最大的容量，如果使用头、尾两个指针，而不使用 Count 变量，如何检测队列是否为空？是否已满？请考虑两种情况：头部在尾部前面，和头部在尾部后面。

11.3 纯函数式实现

仅仅使用一个列表无法满足队列的性能限制。大多数的函数式环境使用单向链表来实现列表，列表的头部操作性能为常数时间，而在尾部需要线性时间 $O(n)$，其中 n 为列表长度。因此入队或者出队中必然有一个操作的性能无法达到要求，如图11.5所示。

(a) DeQueue 为线性时间。

(b) EnQueue 为线性时间。

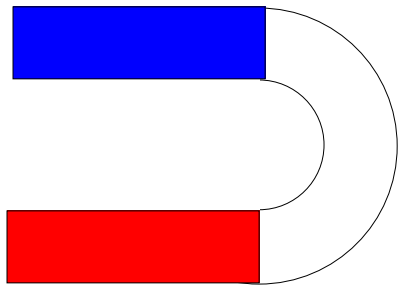
图 11.5: 使用列表，DeQueue 和 EnQueue 无法同时达到常数时间

在纯函数式环境下，我们不能使用变量来记录列表的尾部。因此必须找到一种巧妙的方法，才能实现纯函数式队列。

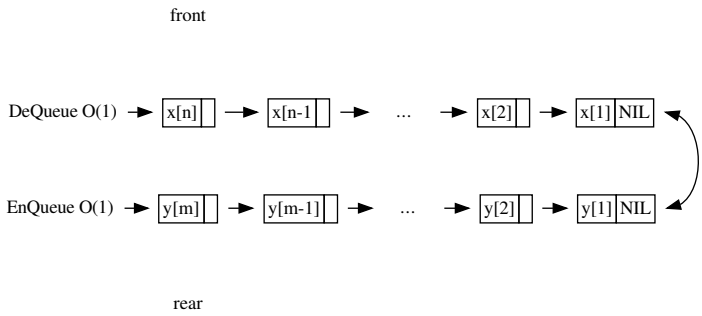
11.3.1 双列表队列

Chris Okasaki 在 [3] 中给出了一个简单直观的函数式实现。思路是使用两个链表来表示一个队列。两个链表“尾对尾”接在一起。形状类似一个马蹄形磁铁，如图11.6所示。

使用两个列表后，我们把新元素加入 rear 列表的头部，性能为常数时间；出队时，我们将元素从 front 列表的头部取走，性能也是常数时间。这样队列的性能要求就都可以满足了。



(a) 马蹄形磁铁



(b) “尾对尾”接在一起的列表

图 11.6: 使用 front 列表和 rear 列表实现的队列形如一个马蹄形磁铁

下面的 Haskell 代码定义了这种双列表（paired-list）队列。

```
type Queue a = ([a], [a])

empty = ([], [])
```

令函数 $front(Q)$ 和 $rear(Q)$ 分别返回 front 和 rear 列表，函数 $Queue(F, R)$ 从两个列表 F 和 R 构造一个队列。入队操作 EnQueue（push）和出队 DeQueue（pop）可以定义如下：

$$push(Q, x) = Queue(front(Q), \{x\} \cup rear(Q)) \tag{11.1}$$

$$pop(Q) = Queue(tail(front(Q)), rear(Q)) \tag{11.2}$$

其中若列表 $X = \{x_1, x_2, ..., x_n\}$，则函数 $tail(X) = \{x_2, x_3, ..., x_n\}$ 为除第一元素外的剩余元素列表。

但是，我们必须解决一个关键问题，经过一系列出队操作后，front 列表有可能变空，而 rear 列表中还有元素。这时如果再进行出队操作将如何处理？一种方案是将 rear 列表反转后替换 front 列表。

为此，每次出队操作后，我们都执行一次平衡检查，记队列 Q 的 front 和 rear 列表分别为 $F = front(Q)$ 和 $R = fear(Q)$。

$$balance(F, R) = \begin{cases} Queue(reverse(R), \phi) & : \quad F = \phi \\ Q & : \quad otherwise \end{cases} \tag{11.3}$$

如果 front 列表不为空，无需任何额外处理；否则如果 front 列表变为空，就用反转的 rear 列表来代替，而新的 rear 列表为空。

这样改动后的入队和出队操作如下：

$$push(Q, x) = balance(F, \{x\} \cup R) \tag{11.4}$$

$$pop(Q) = balance(tail(F), R) \tag{11.5}$$

下面完整的 Haskell 例子程序实现了上述算法。

```
balance :: Queue a → Queue a
balance ([], r) = (reverse r, [])
balance q = q

push :: Queue a → a → Queue a
push (f, r) x = balance (f, x:r)

pop :: Queue a → Queue a
pop ([], _) = error "Empty"
pop (_:f, r) =  balance (f, r)
```

虽然我们仅仅在 front 和 rear 列表的头部进行入队和出队操作，但是性能并不能总保证为常数时间。尽管如此，整体的分摊性能是可以达到常数时间的。将 rear 列表反转所需时间和列表长度成正比，这一步的复杂度为 $O(n)$，其中 $n = |R|$。我们将分担性能的证明作为习题留给读者。

11.3.2 双数组队列——一种对称实现

和双列表相比，存在一个有趣的双数组对称实现。在某些老的编程语言中，例如旧版本的 BASIC，只有数组可用，不能使用指针、或者结构等复合类型来定义链表。尽管可以使用一个额外的数组来记录索引，从而只用数组实现链表，但是还存在更简单的方法来队列，使得分摊性能为常数时间。

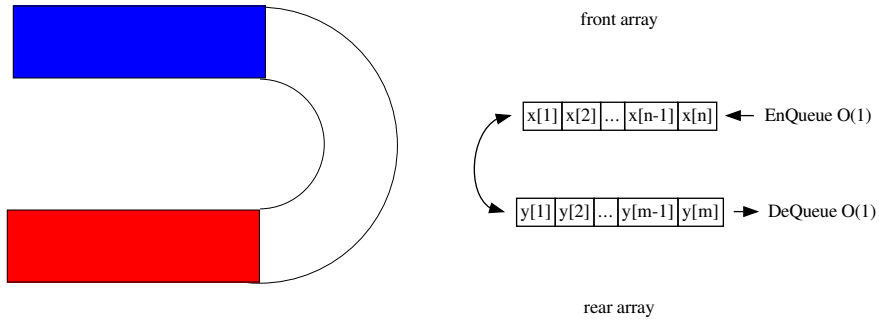
表12.1比较了数组和链表，假设元素个数为 n，各项操作的性能如下：

操作	数组	链表
在头部加入	$O(n)$	$O(1)$
在尾部加入	$O(1)$	$O(n)$
在头部删除	$O(n)$	$O(1)$
在尾部删除	$O(1)$	$O(n)$

表 11.1: 数组和链表各项操作的对比

可以看到，链表在头部的性能为常数时间，而在尾部为线性时间；而数组在尾部操作为常数时间（简单起见，假设空间足够，无需申请），但是在头部操作为线性时间。这是因为在头部插入时，需要将元素依次向后移动以预留出一个空挡，而删除时，需要将后继的元素依次向前移动以填补空挡（参见插入排序一章的介绍）。

上表给出了一个有趣的特性，我们可以利用它设计一个类似双列表的解决方法：将两个数组“头对头”连接起来，形成一个马蹄形的队列，如图11.7所示。



(a) 马蹄形磁铁　　(b) 将两个数组“头对头”连接起来

图 11.7: 使用 front 数组和 rear 数组实现的队列形如一个马蹄形磁铁

下面的 Python 例子程序定义了双数组队列[3]。

```
class Queue:
    def __init__(self):
```

[3]我们省略了旧式 BASIC 代码。在 Python 中，实际是使用了内置的 list 而不是 array。也可参考本书附带的 C/C++ 例子程序，它们使用内置的数组来实现队列。

```
        self.front = []
        self.rear = []

def is_empty(q):
    return q.front == [] and q.rear == []
```

相应的入队操作 PUSH() 和出队操作 POP() 都仅在数组的尾部执行。

function PUSH(Q, x)
  APPEND(REAR(Q), x)

其中过程 APPEND() 将元素 x 加入到数组的尾部，并进行必要的内存申请。有多种内存处理策略，除了重新申请更大的内存，然后复制元素外，还可以设置内存上限，并在用满后报错。

出队操作算法的定义如下：

function POP(Q)
  if FRONT(Q) $= \phi$ **then**
    FRONT(Q) $\leftarrow$ REVERSE(REAR(Q))
    REAR(Q) $\leftarrow \phi$
  $n \leftarrow$ LENGTH(FRONT(Q))
  $x \leftarrow$ FRONT(Q)[n]
  LENGTH(FRONT(Q)) $\leftarrow n - 1$
  return x

简单起见，在删除元素后，我们并未缩小数组的尺寸。可以通过检查数组的长度是否为 0 来判断 front 数组是否为空。这里跳过了这些细节。

下面的 Python 例子程序实现了入队和出队操作。

```
def push(q, x):
    q.rear.append(x)

def pop(q):
    if q.front == []:
        q.rear.reverse()
        (q.front, q.rear) = (q.rear, [])
    return q.front.pop()
```

和 paried-list 队列类似，由于反转数组也是线性时间的，这一实现的分摊性能为常数时间 $O(1)$。

练习 11.3

- 证明双列表队列的分摊性能为常数时间 $O(1)$。
- 证明双数组队列的分摊性能为常数时间 $O(1)$。

11.4 小改进：平衡队列

虽然双列表队列的入队和出队操作的分摊复杂度为常数时间 $O(1)$，但是在最坏情况下的性能很差。例如 front 列表中只有一个元素，然后我们将 n 个元素连续加入队列，这里 n 是一个很大的整数。此时执行一次出队操作就会进入最坏情况。

根据规则，全部 n 个元素都加入了 rear 列表。执行一次弹出操作后，front 列表变为空。于是开始反转 rear 列表，这一操作是线性时间 $O(n)$ 的，和 rear 列表的长度成比例。某些场合下，当 n 很大时，这一时间消耗太大，无法满足要求。

造成这一最坏情况的原因是由于 front 和 rear 列表极不平衡。我们可以修改队列的设计来改进平衡性。例如加入一条平衡限制：

$$|R| \leq |F| \tag{11.6}$$

其中 $R = Rear(Q)$，$F = Front(Q)$ 分别是队列中的前后两个列表。记号 $|L|$ 表示列表 L 的长度。这一条件保证了 rear 列表的长度不大于 front 列表。当条件不满足时，就执行反转操作。

使用这一限制条件，需要频繁获取列表的长度。由于列表本质上是单向链表，因此需要线性时间来得到长度。我们可以将列表的长度缓存起来，并在增减元素时更新长度。这样就可以在常数时间获得列表长度。

下面的 Haskell 例子代码在双列表队列的基础上增加了长度缓存信息。

```
data BalanceQueue a = BQ [a] Int [a] Int
```

只要保持式 (11.6) 一直得到满足，就可以通过检查 front 列表的长度来判断队列是否为空：

$$F = \phi \Leftrightarrow |F| = 0 \tag{11.7}$$

本节余下的部分中，我们一律认为可以在常数时间内获得列表 L 的长度 $|L|$。

入队和出队操作基本和以前一样，我们需要额外传入列表的长度信息，检查平衡条件，如有必要就执行反转操作。

$$push(Q, x) = balance(F, |F|, \{x\} \cup R, |R| + 1) \tag{11.8}$$

$$pop(Q) = balance(tail(F), |F| - 1, R, |R|) \tag{11.9}$$

其中函数 $balance()$ 定义如下：

$$balance(F, |F|, R, |R|) = \begin{cases} Queue(F, |F|, R, |R|) & : & |R| \leq |F| \\ Queue(F \cup reverse(R), |F| + |R|, \phi, 0) & : & otherwise \end{cases} \tag{11.10}$$

这里函数 $Queue()$ 接受四个参数：front 列表和缓存的长度；rear 列表和长度。这些信息用以构造一个双列表队列。

下面的 Haskell 例子程序实现了平衡双列表队列。程序使用了 Haskell 的类型系统来保证实现符合抽象队列的定义。

```
instance Queue BalanceQueue where
    empty = BQ [] 0 [] 0

    isEmpty (BQ _ lenf _ _) = lenf == 0

    -- Amortized O(1) time push
    push (BQ f lenf r lenr) x = balance f lenf (x:r) (lenr + 1)

    -- Amortized O(1) time pop
    pop (BQ (_:f) lenf r lenr) = balance f (lenf - 1) r lenr

    front (BQ (x:_) _ _ _) = x

balance f lenf r lenr
    | lenr ≤ lenf = BQ f lenf r lenr
    | otherwise = BQ (f ++ (reverse r)) (lenf + lenr) [] 0
```

练习 11.4

选择一门命令式语言，实现平衡双数组队列。

11.5 进一步改进：实时队列

改善平衡性后虽然可以避免最差情况，但是反转 rear 列表的的性能仍然是 $O(n)$ 的，其中 $n = |R|$。尽管分摊性能为常数时间，但如果 rear 列表很长，某次操作的性能仍然会很差。在某些实时系统中，我们必须保证在最坏情况下的性能也达到要求。

根据前面的分析，计算的瓶颈在 $F \cup reverse(R)$，当 $|R| > |F|$ 时会发生这一操作。考虑 $|F|$ 和 $|R|$ 都是整数，发生这一操作时，我们有：

$$|R| = |F| + 1 \tag{11.11}$$

F 和 $reverse(R)$ 的结果都是单向链表，连接它们耗时 $O(|F|)$，此外还需要 $O(|R|)$ 时间反转 rear 列表，因此总时间为 $O(n)$，其中 $n = |F| + |R|$。也就是说，和队列中的元素个数成正比。

为了实现实时队列，我们不能一次性计算 $F \cup reverse(R)$。解决策略是将这一耗时的计算分派到各次入队和出队操作中去。这样虽然单次的入队和出队需要做的事情多了，但是可以避免最坏情况下性能退化为线性时间。

逐步反转

我们首先分析一下典型的函数式反转算法。

$$reverse(X) = \begin{cases} \phi & : & X = \phi \\ reverse(X') \cup \{x_1\} & : & otherwise \end{cases} \tag{11.12}$$

其中 $X' = tail(X) = \{x_2, x_3, ...\}$。

若列表为空，则反转结果也是一个空列表。否则，我们取出第一个元素 x_1，将剩余元素 $\{x_2, x_3, ..., x_n\}$ 反转为 $\{x_n, x_{n-1}, .., x_3, x_2\}$，然后再将 x_1 追加到末尾。

但是，这一算法的性能不佳，追加元素到末尾用时和列表长度成正比。因此这一反转操作的性能为 $O(n^2)$。

另外一种实现方式是使用尾递归：

$$reverse(X) = reverse'(X, \phi) \tag{11.13}$$

其中

$$reverse'(X, A) = \begin{cases} A & : & X = \phi \\ reverse'(X', \{x_1\} \cup A) & : & otherwise \end{cases} \tag{11.14}$$

我们称 A 为累积器（accumulator），它不断累积中间结果。任何时候，当调用 $reverse'(X, A)$ 时，X 包含尚未反转的元素，A 包含迄今为止反转完的元素。当第 i 次调用 $reverse'()$ 时，X 和 A 分别包含如下内容：

$$X = \{x_i, x_{i+1}, ..., x_n\} \quad A = \{x_{i-1}, x_{i-2}, ...x_1\}$$

每次递归，如果不是边界情况，我们用常数时间从 X 取出第一个元素；然后将其链结到 A 的前面，这一步同样仅耗时常数时间 $O(1)$。同样的操作重复 n 次，因此这一反转算法是线性时间 $O(n)$ 的。

尾递归 [61][62] 算法可以很容易从一次性计算改为逐步计算。整体过程相当于一系列的状态转换。我们定义一个状态机，包含两种状态：反转状态 S_r 表示正在进行反转（未完成）；完成状态 S_f 表示反转已经结束（完成）。下面的 Haskell 例子程序将状态定义为类型。

```
data State a = | Reverse [a] [a]
               | Done [a]
```

使用这两种状态，我们可以调度（slow-down）函数 $reverse'(X, A)$ 的计算：

$$step(S, X, A) = \begin{cases} (S_f, A) & : & S = S_r \wedge X = \phi \\ (S_r, X', \{x_1\} \cup A) & : & S = S_r \wedge X \neq \phi \end{cases} \tag{11.15}$$

每一步，我们先检查当前的状态，如果状态为 S_r（反转中），但是 X 中已没有剩余元素需要反转，就将状态变换为完成 S_f；否则，我们取出 X 中的第一个元素，将其链结到 A 的前面，接下来与之前不同，我们不再进行递归调用，这一步计算到此结束。当前的状态，以及反转的中间步骤结果 X 和 A 被保存下来，我们可以在以后的任何时候使用这些内容，再调用 $step$ 函数继续反转。

下面是逐步反转的一个例子：

$$
\begin{aligned}
step(S_r, \text{“hello”}, \phi) &= (S_r, \text{“ello”}, \text{“h”}) \\
step(S_r, \text{“ello”}, \text{“h”}) &= (S_r, \text{“llo”}, \text{“eh”}) \\
&... \\
step(S_r, \text{“o”}, \text{“lleh”}) &= (S_r, \phi, \text{“olleh”}) \\
step(S_r, \phi, \text{“olleh”}) &= (S_f, \text{“olleh”})
\end{aligned}
$$

下面的 Haskell 代码描述了同样的例子：

```
step $ Reverse "hello" [] = Reverse "ello" "h"
step $ Reverse "ello" "h" = Reverse "llo" "eh"
...
step $ Reverse "o" "lleh" = Reverse [] "olleh"
step $ Reverse [] "olleh" = Done "olleh"
```

现在我们可以将反转计算逐步分散到入队和出队操作中。但是这仅解决了一半问题。我们需要逐步分解 $F \cup reverse(R)$ 计算。因此接下来需要调度（slow-down）列表的连接操作 $F \cup ...$。连接操作的复杂度为 $O(|F|)$。同样，我们的目标是将它分散到入队和出队操作中去。

逐步连接

实现列表的逐步连接要比逐步反转难度更大。我们可以利用逐步反转的结果。这里有一个小技巧：为了实现 $X \cup Y$，我们可以先将 X 反转为 $\overleftarrow{X}$，然后逐一将 $\overleftarrow{X}$ 中的元素取出，放到 Y 的前面。这和我们前面实现的 $reverse'$ 类似。

$$
\begin{aligned}
X \cup Y &\equiv reverse(reverse(X)) \cup Y \\
&\equiv reverse'(reverse(X), \phi) \cup Y \\
&\equiv reverse'(reverse(X), Y) \\
&\equiv reverse'(\overleftarrow{X}, Y)
\end{aligned} \tag{11.16}
$$

这一事实表明，我们可以增加另一个状态来控制 $step()$ 函数，在 R 反转后，逐步操作 $\overleftarrow{F}$ 实现连接。

整个操作被分解为两个阶段：

1. 同时反转 F 和 R，逐步得到 $\overleftarrow{F} = reverse(F)$ 和 $\overleftarrow{R} = reverse(R)$；
2. 逐步从 $\overleftarrow{F}$ 取出元素，链接到 $\overleftarrow{R}$ 前面。

为此我们定义三种状态：S_r 代表反转；S_c 代表连接；S_f 代表完成。

下面的 Haskell 例子程序定义了这三种状态。

```
data State a = Reverse [a] [a] [a] [a]
             | Concat [a] [a]
             | Done [a]
```

由于我们同时反转 F 和 R，因此反转状态变量带有一对列表和一对累积器。

状态转换按照两个阶段策略进行定义。记 $F = \{f_1, f_2, ...\}$、$F' = tail(F) = \{f_2, f_3, ...\}$、$R = \{r_1, r_2, ...\}$、$R' = tail(R) = \{r_2, r_3, ...\}$。一个状态 $\mathcal{S}$ 包含它的类型 S，可以是 S_r、S_c 和 S_f 的一种。同时状态 $\mathcal{S}$ 中还包含必要的参数，如 F、$\overleftarrow{F}$、X、A 等作为中间结果。状态不同，包含的参数也有所不同。

$$next(\mathcal{S}) = \begin{cases} (S_r, F', \{f_1\} \cup \overleftarrow{F}, R', \{r_1\} \cup \overleftarrow{R}) & : \quad S = S_r \wedge F \neq \phi \wedge R \neq \phi \\ (S_c, \overleftarrow{F}, \{r_1\} \cup \overleftarrow{R}) & : \quad S = S_r \wedge F = \phi \wedge R = \{r_1\} \\ (S_f, A) & : \quad S = S_c \wedge X = \phi \\ (S_c, X', \{x_1\} \cup A) & : \quad S = S_c \wedge X \neq \phi \end{cases} \tag{11.17}$$

相应的 Haskell 程序如下：

```
next (Reverse (x:f) f' (y:r) r') = Reverse f (x:f') r (y:r')
next (Reverse [] f' [y] r') = Concat f' (y:r')
next (Concat 0 _ acc) = Done acc
next (Concat (x:f') acc) = Concat f' (x:acc)
```

接下来我们需要将这些递进的步骤分配到每个出队和入队操作中以实现一个实时 $O(1)$ 的纯函数式队列。

汇总

在给出最终的实现前，我们先来分析一下为了计算 $F \cup reverse(R)$，总共需要多少递进步骤。根据平衡队列的条件，有 $|R| = |F| + 1$。记 $m = |F|$。

由于某次出队或入队操作造成队列不平衡时，我们开始逐步计算 $F \cup reverse(R)$。总共需要 $m + 1$ 步来反转 R，我们同时在这些步骤内完成了对 F 的反转。此后，我们需要再用 $m + 1$ 步来进行连接操作。因此总共花费了 $2m + 2$ 步。

最直观的想法是在每一个出队和入队操作中分配一个递进步骤。但是我们需要回答一个关键问题：在我们完成 $2m + 2$ 步操作之前，队列有没有可能由于接下来的一系列入队和出队操作，再次变得不平衡？

关于这一问题有两个事实，一个是好消息，一个是坏消息。

我们先看好消息。很幸运，在我们花费 $2m + 2$ 步操作计算 $F \cup reverse(R)$ 完成之前，连续的入队操作不可能再次使得队列变得不平衡。因为一旦开始恢复平衡的处理，经过 $2m + 2$ 步后，我们就得到了一个新的 front 列表 $F' = F \cup reverse(R)$。而下一次队列变得不平衡时，我们有：

$$\begin{aligned} |R'| &= |F'| + 1 \\ &= |F| + |R| + 1 \\ &= 2m + 2 \end{aligned} \tag{11.18}$$

也就是说，从上次队列不平衡的时刻算起，即使我们不断持续将新元素入队，以最快的速度再次使得队列不平衡时，$2m + 2$ 步计算恰好已经完成了。此时新的 front

列表被计算出来。我们可以安全地继续计算 $F' \cup reverse(R')$。多亏了前面给出的平衡不变特性（invariant），帮助我们保证了这一点。

但是还有一个坏消息。在 $2m + 2$ 步计算完成前，出队操作可能随时发生。这会产生一个尴尬的情况：我们需要从 front 列表取出元素，但是新的 front 列表 $F' = F \cup reverse(R)$ 尚未计算好。此时没有一个可用的 front 列表。

一种解决方法是在第一阶段计算 $reverse(F)$ 时，另外保存一份此前的 front 列表 F。这样即使连续进行 m 次出队操作，我们仍然是安全的。表 (11.2) 给出了第一阶段逐步计算（同时反转 F 和 R）的某个时刻队列的样子[4]。

保存的 front 列表	进行中的计算	新的 rear 列表
$\{f_i, f_{i+1}, ..., f_M\}$	$(S_r, \tilde{F}, ..., \tilde{R}, ...)$	$\{...\}$
前 $i-1$ 个元素已出队	$\overleftarrow{F}$ 和 $\overleftarrow{R}$ 的中间结果	包含新入队的元素

表 11.2: 前 m 步完成之前的队列中间状态

经过 m 次出队操作，F 的副本已经用光。我们此时刚刚开始进逐步连接的计算阶段。此时如果继续进行出队操作会怎样？

事实上，由于 F 的副本被用光（变成了 ϕ），我们无需再进行连接操作了。这是因为 $F \cup \overleftarrow{R} = \phi \cup \overleftarrow{R} = \overleftarrow{R}$。

这一点告诉我们，在进行连接操作时，我们只需要将 F 中尚未出队的元素连接起来。因为元素从 F 的头部逐一出队，我们可以使用一个计数器（counter）来记录 F 中剩余元素的个数。当开始计算 $F \cup reverse(R)$ 时，计数器为 0，每次反转 F 中的一个元素时，就将计数器加一，表示将来我们需要连接这个元素；每次出队操作，就将计数器减一，表示我们将来可以少连接一个元素。显然在连接操作的每步中，我们也需要递减计数器。当且仅当计数器为 0 的时候，我们无需继续进行连接操作。

根据以上的分析，我们可以给出纯函数式的实时队列的完整实现了。为了简化状态转换，我们可以增加一个空闲状态 S_0。下面的 Haskell 例子程序给出了这一修改过的状态定义。

```
data State a = Empty
             | Reverse Int [a] [a] [a] [a] -- n, f', acc_f' r, acc_r
             | Append Int [a] [a]          -- n, rev_f', acc
             | Done [a] -- result: f ++ reverse r
```

队列的数据结构分为三个部分：front 列表（带有长度信息）；正在计算中的 $F \cup reverse(R)$ 的中间状态；和 rear 列表（带有长度信息）。

下面的 Haskell 例子程序定义了实时队列的数据结构。

```
data RealtimeQueue a = RTQ [a] Int (State a) [a] Int
```

[4]有人会产生疑问，通常复制一个列表需要花费和列表长度成比例的线性时间。这样整个方案就有问题了。实际上，这一线性时间的列表复制根本不会发生。因为在纯函数式的环境下，出队或反转并不“修改”front 列表。但是，如果尝试用双数组实现一个对称解，并且就地修改数组，这一问题就会产生影响。为此，我们需要实现某种 lazy 复制，真正的复制操作并不立即发生，而是在每次反转的递进步骤中一步复制一个元素。具体的实现留给读者作为练习。

空队列包含空的 front 和 rear 列表,以及一个空闲状态 S_0,记为 $Queue(\phi, 0, S_0, \phi, 0)$。根据平衡 invariant 的定义，我们可以通过检查 $|F| = 0$ 与否来判断一个队列是否为空。入队和出队操作修改如下：

$$push(Q, x) = balance(F, |F|, \mathcal{S}, \{x\} \cup R, |R| + 1) \tag{11.19}$$

$$pop(Q) = balance(F', |F| - 1, abort(\mathcal{S}), R, |R|) \tag{11.20}$$

最大的变化在于 $abort()$ 函数。根据前面的分析，在出队时我们递减计数器，这样将来可以少连接一个元素。我们将其定义为撤销操作。我们稍后介绍它的具体实现。

相应的 Haskell 出队和入队操作可以由下面的例子程序给出。

```
push (RTQ f lenf s r lenr) x = balance f lenf s (x:r) (lenr + 1)
pop (RTQ (_:f) lenf s r lenr) = balance f (lenf - 1) (abort s) r lenr
```

函数 $balance()$ 首先检查平衡 invariant,如果违反了,我们需要启动 $F \cup reverse(R)$ 的逐步计算来恢复平衡；否则，我们仅仅执行一步尚未完成的递进计算。

$$balance(F, |F|, \mathcal{S}, R, |R|) = \begin{cases} step(F, |F|, \mathcal{S}, R, |R|) & : \quad |R| \leq |F| \\ step(F, |F| + |R|, (S_r, 0, F, \phi, R, \phi)\phi, 0) & : \quad otherwise \end{cases} \tag{11.21}$$

相应的 Haskell 例子程序如下：

```
balance f lenf s r lenr
    | lenr ≤ lenf =  step f lenf s r lenr
    | otherwise = step f (lenf + lenr) (Reverse 0 f [] r []) [] 0
```

函数 $step()$ 将状态机转换到下一个状态，全部递进计算结束后，状态转换到空闲状态 S_0。

$$step(F, |F|, \mathcal{S}, R, |R|) = \begin{cases} Queue(F', |F|, S_0, R, |R|) & : \quad S' = S_f \\ Queue(F, |F|, \mathcal{S}', R, |R|) & : \quad otherwise \end{cases} \tag{11.22}$$

其中，$\mathcal{S}' = next(\mathcal{S})$，是下一个转换到的状态；$F' = F \cup reverse(R)$ 是递进计算出的新 front 列表。真正的状态转换函数 $next()$ 的实现如下。和前面的定义不同，我们增加了一个计数器 n 来记录还剩余多少个元素需要连接。

$$next(\mathcal{S}) = \begin{cases} (S_r, n+1, F', \{f_1\} \cup \overleftarrow{F}, R', \{r_1\} \cup \overleftarrow{R}) & : \quad S = S_r \wedge F \neq \phi \\ (S_c, n, \overleftarrow{F}, \{r_1\} \cup \overleftarrow{R}) & : \quad S = S_r \wedge F = \phi \\ (S_f, A) & : \quad S = S_c \wedge n = 0 \\ (S_c, n-1, X', \{x_1\} \cup A) & : \quad S = S_c \wedge n \neq 0 \\ \mathcal{S} & : \quad otherwise \end{cases} \tag{11.23}$$

相应的 Haskell 例子程序如下：

```
next (Reverse n (x:f) f' (y:r) r') = Reverse (n+1) f (x:f') r (y:r')
next (Reverse n [] f' [y] r') = Concat n f' (y:r')
next (Concat 0 _ acc) = Done acc
next (Concat n (x:f') acc) = Concat (n-1) f' (x:acc)
next s = s
```

函数 $abort()$ 用于指示状态机，由于发生了出队操作，可以少连接一个元素。

$$abort(\mathcal{S}) = \begin{cases} (S_f, A') & : & S = S_c \wedge n = 0 \\ (S_c, n-1, X'A) & : & S = S_c \wedge n \neq 0 \\ (S_r, n-1, F, \overleftarrow{F}, R, \overleftarrow{R}) & : & S = S_r \\ \mathcal{S} & : & otherwise \end{cases} \tag{11.24}$$

注意当 $n = 0$ 的时候，我们实际上撤销了上一个链接元素的操作，因此返回 A' 而不是 A 作为结果（作为练习，我们请读者来回答这样做的原因）。

下面的 Haskell 例子程序实现了 abort 函数。

```
abort (Concat 0 _ (_:acc)) = Done acc       -- 注意：我们回滚（rollback）了一个元素
abort (Concat n f' acc) = Concat (n-1) f' acc
abort (Reverse n f f' r r') = Reverse (n-1) f f' r r'
abort s = s
```

我们已经接近最终的结果了。但是仍有一个隐藏的问题必须解决：如果将一个元素 x 放入一个空队列，结果会是：

$$Queue(\phi, 1, (S_c, 0, \phi, \{x\}), \phi, 0)$$

若此时立即进行出队操作，就会发生错误！虽然上一次 $F \cup reverse(R)$ 的计算已经结束，但是 front 列表却为空。这是因为还需要额外一步才能从状态 $(S_c, 0, \phi, A)$ 转换到 (S_f, A)。因此需要进一步调整函数 $step()$ 中的 $\mathcal{S}'$ 如下：

$$\mathcal{S}' = \begin{cases} next(next(\mathcal{S})) & : & F = \phi \\ next(\mathcal{S}) & : & otherwise \end{cases} \tag{11.25}$$

下面的 Haskell 例子程序体现了这一修改：

```
step f lenf s r lenr =
    case s' of
      Done f' → RTQ f' lenf Empty r lenr
      s' → RTQ f lenf s' r lenr
    where s' = if null f then next $ next s else next s
```

注意这一算法和 Chris Okasaki 在 [3] 给出的有所不同。Okasaki 的算法每次出队、入队执行两步递进计算，而本章中的算法每次只执行一次。因此计算性能的分布更加均匀。

练习 11.5

- 在 $abort()$ 函数中，当 $n = 0$ 时，为什么需要回滚一个元素？
- 考虑实时队列的对称实现。选择一门命令式语言，用双数组实现实时队列。
- 在脚注中，我们提到，使用就地修改的双数组来实现实时队列时，当开始递进计算反转时，不能一次性复制数组，否则就会将性能降低到线性时间复杂度。请实现一个惰性复制（lazy copy），使得每步反转时我们仅复制一个元素。

11.6 惰性实时队列

实现实时队列的关键在于将耗时的 $F \cup reverse(R)$ 计算分解。惰性求值对于这类问题很有帮助。本节中，我们通过惰性求值来寻找更加简洁的方法。

假设存在一个函数 $rotate()$ 可以逐步计算 $F \cup reverse(R)$。也就是说，使用一个累积器 A，下面的两个函数等价

$$rotate(X, Y, A) \equiv X \cup reverse(Y) \cup A \tag{11.26}$$

其中，我们将 X 初始化为 front 列表 F，Y 初始化为 rear 列表 R，A 初始化为空 ϕ。

开始进行轮转（rotate）的条件和前面一样，即 $|F| + 1 = |R|$。在轮转过程中，我们始终保持 $|X| + 1 = |Y|$ 作为一个不变式成立。

下面我们来推导轮转的实现，显然，最简单的情况如下：

$$rotate(\phi, \{y_1\}, A) = \{y_1\} \cup A \tag{11.27}$$

记 $X = \{x_1, x_2, ...\}$、$Y = \{y_1, y_2, ...\}$；而 $X' = \{x_2, x_3, ...\}$、$Y' = \{y_2, y_3, ...\}$ 是 X 和 Y 除去第一个元素以外的剩余元素。递归情况可以推导如下：

$$\begin{array}{lcll} rotate(X, Y, A) & \equiv & X \cup reverse(Y) \cup A & \text{根据定义 (11.26的定义} \\ & \equiv & \{x_1\} \cup (X' \cup reverse(Y) \cup A) & \cup\text{操作的结合性} \\ & \equiv & \{x_1\} \cup (X' \cup reverse(Y') \cup (\{y_1\} \cup A)) & \text{reverse 的性质和} \cup \text{的结合性} \\ & \equiv & \{x_1\} \cup rotate(X', Y', \{y_1\} \cup A) & \text{根据定义 (11.26)} \end{array} \tag{11.28}$$

归纳上面的两种情况，可以得到最终的轮转算法。

$$rotate(X, Y, A) = \begin{cases} \{y_1\} \cup A & : & X = \phi \\ \{x_1\} \cup rotate(X', Y', \{y_1\} \cup A) & : & otherwise \end{cases} \tag{11.29}$$

如果我们惰性执行 $\cup$ 操作，而不是立即进行链接，也就是说，当出队和入队时才执行 $\cup$，就可以将 $rotate$ 计算自然分摊到出、入队中。

根据这一思路，我们修改双列表队列的定义，将 front 列表变成一个惰性列表，然后将它放入一个流（stream）中 [63]。当某个出、入队操作，造成队列的平衡被破坏，此时有 $|F| + 1 = |R|$，为了恢复平衡，我们开始进行惰性轮转计算。这一惰性计算被作为新的 front 列表 F'，而新的 rear 列表为空 ϕ。我们同时维护一个 F' 的副本作为流。

此后，每当进行出、入队操作，我们就强制流执行一个 $\cup$ 操作。这样流就向前执行一步 $\{x\} \cup F''$，其中 $F'' = tail(F')$。我们丢掉 x，然后用 F'' 替换 F' 作为新的流。

当全部流被计算完毕，就可以开始计算另一个轮转。

为了更好地描述这一思路，我们使用 Scheme/Lisp 给出例子程序。这样可以明确地控制惰性计算。

```
(define (cons-stream a b) (cons a (delay b)))

(define stream-car car)

(define (stream-cdr s) (cdr (force s)))
```

函数 `cons-stream` 从一个元素 x 和列表 L 构造一个惰性列表。它并不对列表 L 求值，求值被推迟到 `stream-cdr` 中进行。延迟求值可以通过 lambda 演算来实现 [63]。

下面的例子程序给出了惰性双列表队列的定义。

```
(define (make-queue f r s)
  (list f r s))

(define (front-lst q) (car q))

(define (rear-lst q) (cadr q))

(define (rots q) (caddr q))
```

一个队列包含三个部分：一个 front 列表，一个 rear 列表和一个代表计算 $F \cup reverse(R)$ 的流。对于空队列，这三个部分全部是 null。

```
(define empty (make-queue '() '() '()))
```

注意，其中的 front 列表实际上是一个惰性流，因此需要使用流相关的操作。例如下面的函数通过检查 front 流来判断队列是否为空。

```
(define (empty? q) (stream-null? (front-lst q)))
```

入队函数和上一节所介绍的基本相同。我们将新加入的元素放到 rear 列表前面，然后检查平衡条件，如果不满足就需要恢复平衡。

$$push(Q, x) = balance(\mathcal{F}, \{x\} \cup R, \mathcal{R}_s) \tag{11.30}$$

其中 $\mathcal{F}$ 是表示 front 列表的惰性流；$\mathcal{R}_s$ 是表示轮转计算的流。相应的 Scheme/Lisp 例子程序如下：

```
(define (push q x)
  (balance (front-lst q) (cons x (rear q)) (rots q)))
```

出队操作和此前相比有一些不同，由于 front 列表实际是一个惰性流，我们需要进行强制求值，其余部分保持不变。

$$pop(Q) = balance(\mathcal{F}', R, \mathcal{R}_s) \tag{11.31}$$

其中 $\mathcal{F}'$ 强制对 $\mathcal{F}$ 进行一次求值，相应的 Scheme/Lisp 例子程序如下：

```
(define (pop q)
  (balance (stream-cdr (front-lst q)) (rear q) (rots q)))
```

为了节省篇幅，我们省略了错误处理（例如对空队列进行出队操作的错误等）。

通过从 front 流中提取元素可以获得队列的头部元素。

```
(define (front q) (stream-car (front-lst q)))
```

平衡函数首先检查代表轮转计算的流，如果已经耗尽，就开始一次新的轮转计算；否则它强制对惰性流进行一次求值，消耗掉其中的一个元素。

$$balance(Q) = \begin{cases} Queue(\mathcal{F}', \phi, \mathcal{F}') & : \quad \mathcal{R}_s = \phi \\ Queue(\mathcal{F}, R, \mathcal{R}_s') & : \quad otherwise \end{cases} \tag{11.32}$$

这里 $\mathcal{F}'$ 被定义来开始一次新的轮转：

$$\mathcal{F}' = rotate(F, R, \phi) \tag{11.33}$$

相应的 Scheme/Lisp 例子程序如下：

```
(define (balance f r s)
  (if (stream-null? s)
      (let ((newf (rotate f r '())))
    (make-queue newf '() newf))
      (make-queue f r (stream-cdr s))))
```

分步递进的轮转函数可以根据我们上面的分析给出实现，如下面的 Scheme/Lisp 例子代码：

```
(define (rotate xs ys acc)
  (if (stream-null? xs)
      (cons-stream (car ys) acc)
      (cons-stream (stream-car xs)
           (rotate (stream-cdr xs) (cdr ys)
               (cons-stream (car ys) acc)))))
```

在 Scheme/Lisp 中，我们可以明确地控制惰性求值。在默认使用惰性求值的编程环境中，例如 Haskell，相应的实现可以非常简洁。

```
data LazyRTQueue a = LQ [a] [a] [a] -- front, rear, f ++ reverse r

instance Queue LazyRTQueue where
    empty = LQ [] [] []

    isEmpty (LQ f _ _) = null f

    -- O(1) time push
    push (LQ f r rot) x = balance f (x:r) rot

    -- O(1) time pop
    pop (LQ (_:f) r rot) = balance f r rot

    front (LQ (x:_) _ _) = x

balance f r [] = let f' = rotate f r [] in LQ f' [] f'
```

```
balance f r (_:rot) = LQ f r rot

rotate [] [y] acc = y:acc
rotate (x:xs) (y:ys) acc = x : rotate xs ys (y:acc)
```

11.7 小结

在第一章的开头，我们曾经说过队列并不像想象中的那么简单。我们此前给出了许多数据结构和算法的命令式实现和函数式实现，函数式的方法通常会更加简洁和直观。但是，还存在许多领域，需要更多的研究工作来寻找相应的函数式解法。队列是一个非常重要的题目，它是很多纯函数式数据结构的基础。

Chris Okasaki 对纯函数式队列进行了集中的研究，给出了许多有益的讨论 [3]。通过解决纯函数式队列，我们可以使用类似的方法实现双向队列（dequeue），同时在队列头部和尾部高效地进行操作。再前进一步，还可以实现序列（sequence）数据结构，支持快速地连接（concatenate），并最终实现随机访问（random access）以模拟命令式环境中的数组。我们将在下一章解释这些细节。

虽然我们没有提到优先队列（priority queue），但它可以很容易地用前面章节中给出的堆（heap）来实现。

练习 11.6

- 使用纯函数的方法，实现双向队列，在头部尾部都支持常数时间 $O(1)$ 的元素添加和删除。
- 选择一门命令式编程语言，利用数组给出双向队列的对称实现。

第十二章　序列，最后一块砖

12.1　简介

本书的第一章把二叉搜索树作为“hello world”的数据结构加以介绍。我们指出，同时给出队列和序列的命令式和函数式实现并不简单。前一章中，我们给出了函数式队列，可以达到和命令式的队列相同的性能。本章中，我们将仔细探讨类似数组的数据结构。

本书中介绍的大部分函数式数据结构往往简洁直观。但是仍然有某些领域，人们尚未发现和命令式实现相当的解法。例如在线性时间内构造后缀树的 Ukkonen 算法、散列表、以及数组。

数组属于命令式环境中的最基本数据结构，它支持通过索引在常数时间 $O(1)$ 内随机访问元素。但是在函数式环境中，只能使用列表作为基本数据结构，我们无法直接获得这样的随机访问性能。

本章中，我们将数组的概念抽象到序列（sequence）。它需要支持下面的特性：

- 可以在序列的头部用常数时间 $O(1)$ 快速插入、删除元素；
- 可以在序列的尾部用常数时间 $O(1)$ 快速插入、删除元素；
- 可以快速（优于线性时间）连接两个序列；
- 可以快速随机访问、更改任何元素；
- 可以快速在任何位置将序列断开；

我们称上述特性为抽象序列性质。显然，即使是命令式环境中的数组（普通数组）也无法同时满足这些要求。

本章中，我们将给出三种解法。首先我们介绍基于二叉树的森林及其数值表示（numeric representation）；接着将介绍可连接列表（concatenate-able list）；最后，我们给出手指树 (finger tree)。

本章大部分的结果来自于 Chris Okasaki 的工作 [3]。

12.2 二叉随机访问列表

二叉树随机访问列表是使用二叉树森林实现的一种随机访问列表。在进一步分析它的实现前，我们首先对比一下列表和数组在各个方面的差异。

12.2.1 普通数组和列表

表12.1列出了普通数组和单向链表的性能对比，我们可以看到它们在不同情况下的表现。

操作	数组	链表
在头部操作	$O(n)$	$O(1)$
在尾部操作	$O(1)$	$O(n)$
随机访问	$O(1)$	平均 $O(n)$
在给定位置删除	平均 $O(n)$	$O(1)$
连接	$O(n_2)$	$O(n_1)$

表 12.1: 普通数组和单向链表的对比

由于持有表头，所以在链表的头部进行插入和删除只需要常数时间；但是我们需要遍历到链表的末尾来进行尾部的删除和追加；给定位置 i，我们首先需要前进 i 步以到达第 i 个元素的位置。此后，只需要做指针的改动就可以在常数时间内完成元素的删除。为了连接两个链表，我们需要遍历到第一个链表的末尾，然后链接到第二个链表的表头。因此所用时间和第一个链表的长度成比例。

而对于数组，我们必须将第一个 cell 空出以在头部插入一个新元素；删除第一个元素后，我们需要将第一个 cell 释放。这两个操作都需要对后续元素进行逐一移动，因此花费线性时间。反之，在数组的尾部进行操作则很简单，只需要常数时间。数组支持常数时间访问第 i 个元素；但是将第 i 个元素删除则需要将后面的元素逐一向前移动。为了连接两个数组，我们需要将第二个数组的元素全部复制到第一个数组的后面（忽略内存重新分配的细节），因此所用时间和第二个数组的长度成比例。

在二项式堆的章节中，我们给出了森林的概念，森林是若干树的列表。它的好处是，给定任何非负整数 n，将其表达为二进制，我们就知道需要多少棵二项式树来存储 n 个元素。每个值为 1 的二进制位代表一棵二项式树，树的 rank 为对应的第几个二进制位。我们可以得到进一步的结论：对于含有 n 个节点的二项式堆，给定任何索引 $1 < i < n$，我们都可以快速地在堆中定位到保存第 i 个节点的二项式树。

12.2.2 使用森林表示序列

可以使用完全二叉树的森林来实现随机访问序列。图12.1展示了如何将使用若干完全二叉树来管理序列。

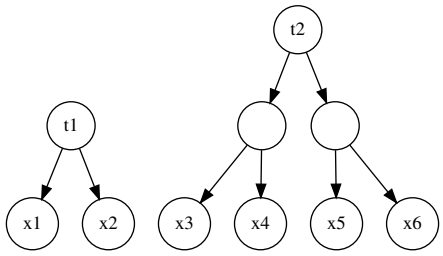


图 12.1: 使用森林来表示含有 6 个元素的序列

图中使用了两棵树 t_1 和 t_2 来表示序列 $\{x_1, x_2, x_3, x_4, x_5, x_6\}$。二叉树 t_1 的大小为 2。前两个元素 $\{x_1, x_2\}$ 存储在 t_1 的叶子中；二叉树 t_2 的大小为 4，接下来的 4 个元素 $\{x_3, x_4, x_5, x_6\}$ 保存在 t_2 中。

对于完全二叉树，我们定义只含有一个叶子的树深度为 0。将深度为 $i+1$ 的树记为 t_i。显然，t_i 含有 2^i 个叶子。

对于含有任意 n 个元素的序列，我们可以将其转换为一组这样的完全二叉树森林。首先我们将 n 表示为如下的二进制数。

$$n = 2^0 e_0 + 2^1 e_1 + ... + 2^m e_m \tag{12.1}$$

其中 e_i 的值为 1 或 0，即 $n = (e_m e_{m-1} ... e_1 e_0)_2$。如果 $e_i \neq 0$，就需要一棵大小为 2^i 的完全二叉树。例如图12.1中，序列的长度为 6，写成二进制为 $(110)_2$。最低位是 0，因此我们不需要大小为 1 的树；第 2 位为 1，因此需要一棵大小为 2 的树；最高位也是 1，因此需要一棵深度为 3，大小为 4 的树。

这一方法将序列 $\{x_1, x_2, ..., x_n\}$ 表示为一个树的列表 $\{t_0, t_1, ..., t_m\}$，其中若 $e_i = 0$，则 t_i 为空，否则，若 $e_i = 1$，则 t_i 为一棵完全二叉树。我们称这一树的列表为二叉随机访问列表（Binary Random Access List）[3]。

我们可以复用二叉树的定义。下面的 Haskell 例子程序定义了树和二叉随机访问列表。

```
data Tree a = Leaf a
            | Node Int (Tree a) (Tree a)  -- size, left, right

type BRAList a = [Tree a]
```

和标准二叉树定义相比，我们增加了 size 的信息。这样能够避免每次都递归计算 size，可以用常数时间获取树的大小。

```
size (Leaf _) = 1
size (Node sz _ _) = sz
```

12.2.3 在序列的头部插入

使用森林表示序列可以高效实现许多操作。例如，将新元素 y 插入到序列的前面可以实现如下：

1. 创建一个只含有一个叶子节点 y 的树 t'；

2. 检查森林中的第一棵树，比较它和 t' 的 size 大小，如果它的 size 大于 t' 的，就将 t' 作为森林中的第一棵树，由于森林本质是一个树的链表，在表头插入 t' 只需要常数时间 $O(1)$；

3. 否则，若森林中的第一棵树的 size 和 t' 相等，记森林中的这棵树为 t_i，可以通过将 t_i 和 t' 链接起来，形成一棵新树 t'_{i+1}，t_i 和 t' 分别为这棵新树的左右子树。然后我们递归地将 t'_{i+1} 插入到森林中。

图12.2描述了将元素 $x_1, x_2, ..., x_6$ 依次插入到空列表的情况。

由于森林中最多包含 m 棵树，m 的大小为 $O(\lg n)$，因此在头部插入的算法最坏情况下的性能为 $O(\lg n)$。稍后我们会证明分摊性能为 $O(1)$。

接下来我们将上述算法形式化。定义将元素插入到序列头部的函数为 $insert(S, x)$。

$$insert(S, x) = insertTree(S, leaf(x)) \tag{12.2}$$

这一函数从元素 x 构造一棵只有一个叶子节点的树，然后调用 $insertTree$ 将树插入到森林中。若森林不为空，记 $F = \{t_1, t_2, ...\}$，$F' = \{t_2, t_3, ...\}$ 表示森林中除第一棵树外的剩余部分。

$$insertTree(F, t) = \begin{cases} \{t\} & : & F = \phi \\ \{t\} \cup F & : & size(t) < size(t_1) \\ insertTree(F', link(t, t_1)) & : & otherwise \end{cases} \tag{12.3}$$

其中函数 $link(t_1, t_2)$ 从两棵 size 相同的较小的树构造一棵新树。函数 $tree(s, t_1, t_2)$ 用于构造一棵树，size 为 s，左右子树分别为 t_1 和 t_2。树的链接可以实现如下：

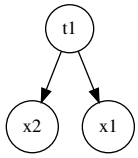
$$link(t_1, t_2) = tree(size(t_1) + size(t_2), t_1, t_2) \tag{12.4}$$

下面的 Haskell 例子程序实现了在表头插入元素的算法。

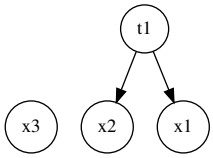
```
cons :: a → BRAList a → BRAList a
cons x ts = insertTree ts (Leaf x)

insertTree :: BRAList a → Tree a → BRAList a
insertTree [] t = [t]
insertTree (t':ts) t = if size t < size t' then  t:t':ts
                       else insertTree ts (link t t')

-- Precondition: rank t1 = rank t2
link :: Tree a → Tree a → Tree a
link t1 t2 = Node (size t1 + size t2) t1 t2
```

这段代码中，我们使用了 Lisp 中的命名传统，把向列表头部插入元素称为“cons”。

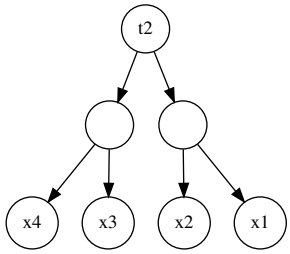
(a) 包含一个元素 x_1 的叶子。



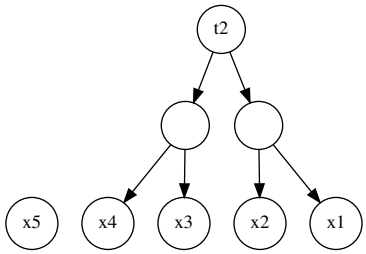
(b) 插入 x_2。进行链接，产生一棵高度为 1 的树。



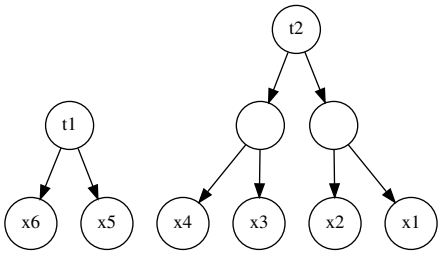
(c) 插入 x_3。结果包含两棵树 t_1 和 t_2。



(d) 插入 x_4。首先将两个叶子链接成一棵二叉树，然后再次链接，产生一棵高度为 2 的树。



(e) 插入 x_5。森林中包含两棵树，一棵仅含有一个叶子节点（t_0），另一棵为 t_2。



(f) 插入 x_6。将两个叶子链接成 t_1。

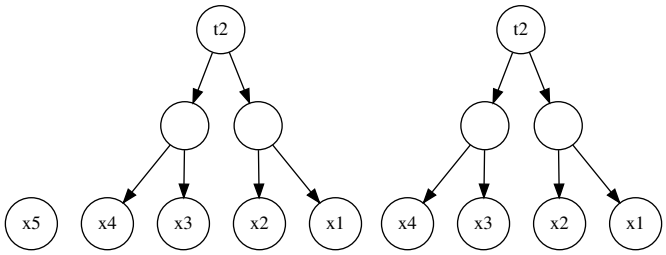
图 12.2: 将元素插入到空列表的步骤

从序列头部删除元素

类似地，我们可以实现“cons”的逆运算，从序列的头部删除元素。

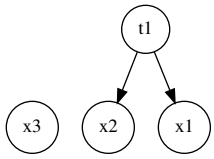
- 如果森林中的第一棵树只含有一个叶子节点，将这棵大小为 1 的树删除；
- 否则，将第一棵树的两个子树拆分，然后递归地将第一棵子树继续拆分直到获得一棵只有一个叶子节点的树。

图12.3给出了从序列头部删除元素的步骤。



(a) 含有 5 个元素的序列

(b) 删除 x_5，大小为 1，只有一个叶子节点的树被删除。



(c) 删除 x_4，由于不是仅包含一个叶子节点的树，它首先被拆分成大小为 2 的两棵子树。然后第一棵子树继续被拆分成两个叶子。此后含有 x_4 的第一个叶子被删除。森林中剩下两棵树，一棵是含有 x_3 的子树，另外一棵大小为 2，包含元素 x_2, x_1。

图 12.3: 从头部删除元素的步骤

简单起见，假设序列不为空，我们可以忽略从空序列中删除元素的错误处理。上述算法可以表示为下面的表达式。记森林为 $F = \{t_1, t_2, ...\}$，除去第一棵树后的剩余部分为 $F' = \{t_2, t_3, ...\}$。

$$extractTree(F) = \begin{cases} (t_1, F') & : \quad t_1 \quad \text{is leaf} \\ extractTree(\{t_l, t_r\} \cup F') & : \quad otherwise \end{cases} \tag{12.5}$$

其中 $\{t_l, t_r\} = unlink(t_1)$，为 t_1 的两棵子树。

下面的 Haskell 程序实现了这一算法。

```
extractTree (t@(Leaf x):ts) = (t, ts)
extractTree (t@(Node _ t1 t2):ts) = extractTree (t1:t2:ts)
```

使用这一定义，可以很容易地给出 $head$ 和 $tail$ 函数，前者返回序列中的第一个元素，后者返回剩余元素。

$$head(S) = key(first(extractTree(S))) \tag{12.6}$$

$$tail(S) = second(extractTree(S)) \tag{12.7}$$

其中函数 $first$ 返回一对元素（亦称为 tuple）中的前一个元素；$second$ 返回后一个元素。函数 key 用以访问叶子节点中存储的元素。下面的 Haskell 例子程序实现这两个函数。

```
head' ts = x where (Leaf x, _) = extractTree ts
tail' = snd ∘ extractTree
```

为了和 Haskell 的标准库中定义的 `head` 和 `tail` 区别，我们加上了“'”号（另外一种方法是通过隐藏 import 不导入标准库中的定义，我们忽略这些和语言相关的特定细节）。

随机访问元素

森林中的树实际上将元素划分为大小不同的块进行管理，给定任意索引，可以很容易地定位到保存此元素的树。然后在树中进行一次查找就可以得到结果。因为所有的树都是二叉树（确切地说是完全二叉树），所以树中的查找实际上是二分查找，性能和树的大小成对数比例。进行随机访问时，相比在链表中进行线性查找要快得多。

给定索引 i 和序列 S，序列使用由树组成的森林来表示，随机访问算法可以描述如下[1]：

1. 比较 i 和森林中第一棵树 T_1 的 size 的大小，若 i 小于等于 size，则元素在 T_1 中，接下来在 T_1 中查找；
2. 否则，从 i 中减去 T_1 的 size，然后在森林中剩余的树中重复前面的步骤。

这一算法可以形式化为下面的定义。

$$get(S, i) = \begin{cases} lookupTree(T_1, i) & : \quad i \leq |T_1| \\ get(S', i - |T_1|) & : \quad otherwise \end{cases} \tag{12.8}$$

其中 $|T| = size(T)$，而 $S' = \{T_2, T_3, ...\}$ 为森林中除第一棵树以外的剩余部分。这里我们没有进行越界检查和错误处理，这些可以留给读者作为练习。

函数 $lookupTree$ 是一个二分查找算法，如果 i 等于 1，我们返回树的根节点，否则，我们将树对半拆分，如果 i 小于拆分后树的 size，就递归地在左子树中查找，否则就递归地在右子树中查找。

$$lookupTree(T, i) = \begin{cases} root(T) & : \quad i = 1 \\ lookupTree(left(T)) & : \quad i \leq \lfloor \frac{|T|}{2} \rfloor \\ lookupTree(right(T)) & : \quad otherwise \end{cases} \tag{12.9}$$

[1]按照传统，在进行算法描述时，索引 i 从 1 开始；在大多数编程语言中，索引从 0 开始。

其中函数 $left$ 返回 T 的左子树 T_l，而 $right$ 返回右子树 T_r。

下面的 Haskell 例子程序实现了相应的算法。

```
getAt (t:ts) i = if i < size t then lookupTree t i
                 else getAt ts (i - size t)

lookupTree (Leaf x) 0 = x
lookupTree (Node sz t1 t2) i = if i < sz `div` 2 then lookupTree t1 i
                               else lookupTree t2 (i - sz `div` 2)
```

图12.4描述了在一个长度为 6 的序列中查找第 4 个元素的步骤。首先检查第一棵树，由于大小为 2，小于 4，所以继续检查第二棵树，同时将索引更新为 $i' = 4 - 2$。即查找森林中剩余部分的第 2 个元素。由于接下来的树大小为 4，大于 2，所以待查找的元素就在这棵树中。因为新索引为 2，它不大于对半拆分的子树尺寸 4/2=2，所以接下来需要检查左子树，然后检查右侧的孙子分支，最终得到要访问的元素。

使用类似的思路，我们可以修改任意位置 i 的元素。首先比较森林中第一棵树 T_1 的 size 和 i 的大小，若小于 i，待修改的元素不在第一棵树中。我们递归地检查森林中的下一棵树，将它的 size 和 $i - |T_1|$ 比较，其中 $|T_1|$ 是第一棵树的 size。否则，若这一 size 大于等于 i，待修改的元素在树中，我们递归地拆分树，直到获得叶子节点，然后将节点中的元素替换为新元素。

$$set(S, i, x) = \begin{cases} \{updateTree(T_1, i, x)\} \cup S' & : \quad i < |T_1| \\ \{T_1\} \cup set(S', i - |T_1|, x) & : \quad otherwise \end{cases} \tag{12.10}$$

其中 $S' = \{T_2, T_3, ...\}$ 是森林中除第一棵树外的剩余部分。

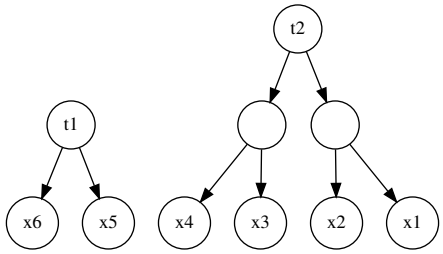
函数 $setTree(T, i, x)$ 执行树搜索，并将第 i 个元素替换为 x。

$$setTree(T, i, x) = \begin{cases} leaf(x) & : \quad i = 0 \wedge |T| = 1 \\ tree(|T|, setTree(T_l, i, x), T_r) & : \quad i < \lfloor \frac{|T|}{2} \rfloor \\ tree(|T|, T_l, setTree(T_r, i - \lfloor \frac{|T|}{2} \rfloor, x)) & : \quad otherwise \end{cases} \tag{12.11}$$

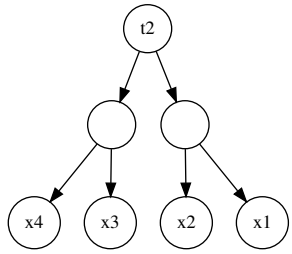
其中 T_l 和 T_r 分别为 T 的左右子树。下面的 Haskell 例子程序实现了这一算法。

```
setAt :: BRAList a → Int → a → BRAList a
setAt (t:ts) i x = if i < size t then (updateTree t i x):ts
                   else t:setAt ts (i-size t) x

updateTree :: Tree a → Int → a → Tree a
updateTree (Leaf _) 0 x = Leaf x
updateTree (Node sz t1 t2) i x =
    if i < sz `div` 2 then Node sz (updateTree t1 i x) t2
    else Node sz t1 (updateTree t2 (i - sz `div` 2) x)
```

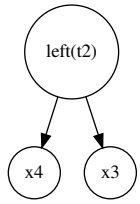
根据完全二叉树的性质，对于含有 n 个元素，用二叉随机访问列表表示的序列，森林中树木的棵数为 $O(\lg n)$。因此任意给定索引 i，最多需要 $O(\lg n)$ 时间来定位到树。接下来的搜索和树的高度成正比，最多也是 $O(\lg n)$。因此随机访问的总体性能为 $O(\lg n)$。



(a) $getAt(S, 4))$, $4 > size(t_1) = 2$



(b) $getAt(S', 4 - 2) \Rightarrow lookupTree(t_2, 2)$



(c) $2 \leq \lfloor size(t_2)/2 \rfloor \Rightarrow lookupTree(left(t_2), 2)$

(d) $lookupTree(right(left(t_2)), 1)$, x_3 被返回

图 12.4: 在序列中访问第 4 个元素的步骤

练习 12.1

1. 本节给出的随机访问算法没有处理索引越界的错误情况。选择一门编程语言，修改算法，实现错误处理。

2. 也可以用命令式方式实现二叉随机访问列表，提供在序列头部的快速操作。随机访问可以通过两个步骤实现：首先定位到树，然后利用数组的常数时间随机访问能力。选择一门命令式语言实现这一思路。

12.3 二叉随机访问列表的数字表示（Numeric representation）

在前一节，我们提到对于任意长度为 n 的序列，可以将 n 表示为二进制形式 $n = 2^0 e_0 + 2^1 e_1 + ... + 2^m e_m$，其中 e_i 为第 i 位，值为 1 或者 0。若 $e_i \neq 0$，则存在一棵大小为 2^i 的完全二叉树。

这一事实反映了 n 的二进制形式和森林之间存在明确的关系。向序列的头部插入元素，类似于将二进制数增加 1；而从序列的头部删除元素类似于将二进制数减少 1。我们称这种关系为numeric representation[3]。

为了将二叉随机访问列表用二进制数字表示，可以为每一个二进制位定义两个状态。状态 $Zero$ 表示不存在对应此二进制位大小的树，而 One 表示森林中存在一棵对应于此二进制位大小的树。如果状态为 One，我们可以将对应的树附加到状态上。

下面的 Haskell 例子程序定义了这样状态。

```
data Digit a = Zero
             | One (Tree a)

type RAList a = [Digit a]
```

我们重用了完全二叉树的定义，并把它附加到状态 One 上。同时将树的大小信息也加以缓存。

定义了数字（digit）后，一个森林就可以按照包含若干数字的列表来处理。我们首先看如何将新元素的插入操作实现为二进制数的增加。设函数 $one(t)$ 创建一个 One 的状态，并将树 t 附加到这个状态上。函数 $getTree(s)$ 从状态 s 中获取树。序列 S 是一个列表，包含若干表示状态的数字，记为 $S = \{s_1, s_2, ...\}$。S' 为除第一个状态外的剩余部分。

$$insertTree(S, t) = \begin{cases} \{one(t)\} & : & S = \phi \\ \{one(t)\} \cup S' & : & s_1 = Zero \\ \{Zero\} \cup insertTree(S', link(t, getTree(s_1))) & : & otherwise \end{cases} \tag{12.12}$$

将一棵新树 t 插入到一个用二进制数字序列表示的森林 S 时，若森林为空，我们创建一个状态 One，将待插入的树附加到状态上。这个状态是二进制数中的唯一一位。相当于二进制加法 $0+1=1$。

否则，如果森林不为空，我们需要检查二进制数的第一位，如果第一个数字是 $Zero$，我们创建一个状态 One，附加上待插入的树，然后用这个新创建的 One 状态替换掉 $Zero$ 状态。这相当于二进制加法 $(...digits...0)_2+1=(...digits...1)_2$。例如 $6+1=(110)_2+1=(111)_2=7$。

最后一种情况是二进制数的第一位数字是 One，这里我们假设待插入的树 t 和状态 One 中附加的树具有相同的 size。这一点可以通过插入过程得到保证，我们总是从一个叶子开始插入,然后待插入的树的大小逐渐增长,呈一个序列 $1,2,4,...,2^i,...$。此时，我们将两棵树链接成一棵更大的树，然后递归地将链接结果插入到剩余的数字中。而之前的 One 状态，被替换为一个 $Zero$ 状态。这相当于二进制加法 $(...digits...1)_2+1=(...digits'...0)_2$，其中 $(...digits'...)_2=(...digits...)_2+1$。例如 $7+1=(111)_2+1=(1000)_2=8$。

下面的 Haskell 例子程序实现了这一算法。

```
insertTree :: RAList a → Tree a → RAList a
insertTree [] t = [One t]
insertTree (Zero:ts) t = One t : ts
insertTree (One t' :ts) t = Zero : insertTree ts (link t t')
```

其他函数，包括 $link()$、$cons()$ 等和此前的定义一样。

接下来我们解释如何用二进制数的减法来表示从序列的头部删除元素。如果序列只含有唯一的状态 One，且状态上附加的树只有一个叶子。删除后序列变为空。这相当于二进制减法 $1-1=0$。

否则，我们检查序列中的第一个数字，如果是 One，它将被替换为 $Zero$，表示森林中的这棵树将被删除。这相当于二进制减法 $(...digits...1)_2-1=(...digits...0)_2$。例如 $7-1=(111)_2-1=(110)_2=6$;

如果序列中的第一个数字是 $Zero$，我们需要向后继的数字借位来进行删除。我们递归地从剩余的数字中抽取树，将其分拆成两棵子树。$Zero$ 状态将被替换成 One 状态，并将此前分拆出的右子树附加到状态上，而删除掉左子树。这相当于二进制减法 $(...digits...0)_2-1=(...digits'...1)_2$，其中 $(...digits'...)_2=(...digits...)_2-1$。例如 $4-1=(100)_2-1=(11)_2=3$。下面的定义给出了删除算法。

$$extractTree(S)=\begin{cases}(t,\phi) & : & S=\{one(t)\}\\ (t,S') & : & s_1=one(t)\\ (t_l,\{one(t_r)\}\cup S'') & : & otherwise\end{cases} \tag{12.13}$$

其中 $(t',S'')=extractTree(S')$，t_l 和 t_r 分别是 t' 的左右子树。其他函数，包括 $head$ 和 $tail$ 的定义和此前一样。

使用数字表示二叉随机访问列表并没有改变复杂度，Okasaki 在 [64] 中给出了

详细地解释。作为例子，我们使用聚合（aggregation）法，来分析在头部插入的平均（或称分摊）复杂度。

考虑依次向一个空的二叉随机访问列表插入 $n = 2^m$ 个元素的过程。森林的二进制表示可以列成表12.2：

i	森林 (MSB ... LSB)
0	0, 0, ..., 0, 0
1	0, 0, ..., 0, 1
2	0, 0, ..., 1, 0
3	0, 0, ..., 1, 1
...	...
$2^m - 1$	1, 1, ..., 1, 1
2^m	1, 0, 0, ..., 0, 0
位变化的次数	1, 1, 2, ... 2^{m-1}, 2^m

表 12.2: 插入 2^m 个元素的过程中森林的二进制表示

森林对应的 LSB（最低位）每次在插入新元素时都变化，它总共需要 2^m 单位次计算；接下来的一位每隔一次变化，执行一次树的链接操作。总共需要 2^{m-1} 单位次计算；森林中对应 MSB（最高位）的前一位总共只变化一次，它将此前所有的树链接成一棵更大的、森林中唯一的树，这发生在插入过程中的正中间。当最后一个元素插入后，MSB 变化成 1。

将所有的这些计算次数相加，我们得到 $T = 1+1+2+4+...+2^{m-1}+2^m = 2^{m+1}$。因此平均下来每次插入操作的计算耗时：

$$O(T/N) = O(\frac{2^{m+1}}{2^m}) = O(1) \tag{12.14}$$

这证明了插入算法的分摊复杂度为常数时间 $O(1)$。删除算法复杂度的证明留给读者作为练习。

12.3.1 命令式二叉随机访问列表

使用二叉树实现命令式二叉随机访问列表非常简单，递归可以通过在循环中修改当前的树进行消除。我们将此作为练习留给读者。本节中，我们给出一些不同的命令式实现，但基本思路仍然是使用数值表示法。

回顾一下二叉堆一章中的内容。二叉堆可以通过隐式的数组来实现。我们可以借鉴类似的方法，用只含有一个元素的数组代表叶子；用含有 2 个元素的数组代表高度为 1 的二叉树；用含有 2^m 个元素的数组代表高度为 m 的完全二叉树。

这样做的好处是，我们可以通过索引快速访问任何元素，而无需进行分而治之的树查找。代价是树的链接操作被替换成了耗时的数组复制。

下面的 ANSI C 例子代码定义了二叉树森林。

```
#define M sizeof(int) * 8
typedef int Key;

struct List {
    int n;
    Key* tree[M];
};
```

其中 n 是森林中存储的元素个数。当然，我们也可以通过使用动态数组来避免限制树的总数。例如下面的 ISO C++ 例子程序。

```
template<typename Key>
struct List {
    int n;
    vector<vector<key> > tree;
};
```

简单起见，我们使用 ANSI C 作为例子。

首先回顾一下插入过程，若第一棵树为空（一个状态为 $Zero$ 的数字），只需要将第一棵树变成一棵含有一个叶子节点的树，将待插入元素放入其中；否则，插入将引发树的链接，这一过程是递归的，直到某一个位置（digit），这个位置上对应的树为空。数值表示告诉我们，如果第一棵、第二棵、……、第 $i-1$ 棵树都存在，而第 i 棵树为空，结果会构造一棵大小为 2^i 的树，待插入的元素，和所有此前的元素都存储在这棵树中。而位置 i 之后的所有树都保持不变。

怎样能高效地定位到位置 i 呢？如果使用二进制数来代表含有 n 个元素的森林，当插入一个新元素后，n 增长到 $n+1$。比较 n 和 $n+1$ 的二进制形式可以发现，所有 i 之前的位都从 1 变到 0，而第 i 位从 0 变成 1，所有 i 之后的位都保持不变。我们可以用位运算异或（$\oplus$）来检测到这一位，算法如下：

function NUMBER-OF-BITS(n)
  $i \leftarrow 0$
  while $\lfloor \frac{n}{2} \rfloor \neq 0$ **do**
    $n \leftarrow \lfloor \frac{n}{2} \rfloor$
    $i \leftarrow i + 1$
  return i

$i \leftarrow$ NUMBER-OF-BITS($n \oplus (n+1)$)

也可以通过移位运算来计算一个二进制数中 1 的个数，如下面的 ANSI C 例子程序。

```
int nbits(int n) {
    int i=0;
    while(n >>= 1)
      ++i;
    return i;
```

```
}
```

因此，命令式插入算法可以这样实现：首先定位到从 0 翻转成 1 的位 i，然后创建一个大小为 2^i 的数组，用以代表相应的完全二叉树，最后将待插入元素和此位之前所有的内容移动到这一数组中。

function INSERT(L, x)
  $i \leftarrow$ NUMBER-OF-BITS($n \oplus (n+1)$)
  TREE(L)$[i+1] \leftarrow$ CREATE-ARRAY(2^i)
  $l \leftarrow 1$
  TREE(L)$[i+1][l] \leftarrow x$
  for $j \in [1, i]$ **do**
    for $k \in [1, 2^j]$ **do**
      $l \leftarrow l + 1$
      TREE(L)$[i+1][l] \leftarrow$ TREE(L)$[j][k]$
    TREE(L)$[j] \leftarrow$ NIL
  SIZE(L) $\leftarrow$ SIZE(L) $+ 1$
  return L

对应的 ANSI C 例子程序如下。

```
struct List insert(struct List a, Key x) {
    int i, j, sz;
    Key* xs;
    i = nbits((a.n + 1) ^ a.n);
    xs = a.tree[i] = (Key*)malloc(sizeof(Key) * (1 << i));
    for(j = 0, *xs++ = x, sz = 1; j < i; ++j, sz <<= 1) {
        memcpy((void*)xs, (void*)a.tree[j], sizeof(Key) * (sz));
        xs += sz;
        free(a.tree[j]);
        a.tree[j] = NULL;
    }
    ++a.n;
    return a;
}
```

但是这一方法的性能在理论上不如前面的好。这是因为原本常数时间的链接操作，下降成了线性时间的数组复制。

我们可以再次使用聚合法分析分摊性能。列表使用由数组表示的二叉树森林来实现，连续向一个空列表插入 $n = 2^m$ 个元素，森林对应的二进制数变化和前面一样，但是每一个二进制位翻转时的计算量和此前有所不同，如表12.3所示：

森林的 LSB 每次插入新元素都会变化，但是只有在从 0 到 1 的变化时会创建含有一个叶子的树并进行复制，因此成本是 $\frac{n}{2}$ 个计算单位，为 2^{m-1}；下一位翻转的次数为 LSB 的一半，每当翻转成 1 时，就把待插入元素和第一棵树中的内容复制到第二棵树中。因此翻转一次的成本为 2 个计算单位，而不是 1 个；对于 MSB，它在最

i	森林 (MSB ... LSB)
0	0, 0, ..., 0, 0
1	0, 0, ..., 0, 1
2	0, 0, ..., 1, 0
3	0, 0, ..., 1, 1
...	...
$2^m - 1$	1, 1, ..., 1, 1
2^m	1, 0, 0, ..., 0, 0
位变化的计算量	1×2^m, $1 \times 2^{m-1}$, $2 \times 2^{m-2}$, ... $2^{m-2} \times 2$, $2^{m-1} \times 1$

表 12.3: 插入 2^m 个元素的过程中森林的二进制表示

后一次翻转成 1，但是成本是将所有此前的树中的元素复制到大小为 2^m 的数组中。

将全部计算成本相加并除以插入次数 n，就得到了分摊性能：

$$\begin{aligned} O(T/N) &= O(\frac{1 \times 2^m + 1 \times 2^{m-1} + 2 \times 2^{m-2} + ... + 2^{m-1} \times 1}{2^m}) \\ &= O(1 + \frac{m}{2}) \\ &= O(m) \end{aligned} \tag{12.15}$$

因为 $m = O(\lg n)$，所以分摊性能从常数时间下降为对数时间。但是这仍然比普通的数组插入要快，普通数组插入的平均性能为 $O(n)$。

由于使用数组的索引，随机访问的性能要比此前的树查找更快。

function GET(L, i)
  for each $t \in$ TREES(L) **do**
    if $t \neq$ NIL **then**
      if $i \leq$ SIZE(t) **then**
        return t[i]
      else
        $i \leftarrow i -$ SIZE(t)

简单起见，我们省略了索引越界的错误处理，相应的 ANSI C 例子程序如下：

```
Key get(struct List a, int i) {
    int j, sz;
    for(j = 0, sz = 1; j < M; ++j, sz <<= 1)
        if(a.tree[j]) {
            if(i < sz)
                break;
            i -= sz;
        }
    return a.tree[j][i];
}
```

命令式的删除和随机访问修改元素的算法留给读者作为练习。

练习 12.2

1. 选择一门语言，实现数值表示的随机访问算法，包括查找和修改指定位置的元素。
2. 使用聚合法，证明删除算法的分摊复杂度为常数时间 $O(1)$。
3. 选择一门命令式语言，设计并实现用数组表示的二叉随机访问列表。

12.4 命令式双数组列表（paired-array list）

在前面的章节中，我们给出过一个用双数组（paired-array）实现的队列。在首尾两端都支持快速的操作。由于数组具备快速随机访问的性质，这一方法也可以用来实现命令式的列表。

12.4.1 定义

图12.5给出了双数组列表的结构。两个数组按照头对头的方式连接起来。在列表的头部插入元素时，新元素被添加到 front 数组的后尾；向列表的尾部插入元素时，新元素被添加到 rear 数组的末尾。

图 12.5: 一个双数组列表，包含两个头对头连接起来的数组

下面的 ISO C++ 例子程序定义了这样的一个数据结构。

```
template<typename Key>
struct List {
    int n, m;
    vector<Key> front;
    vector<Key> rear;

    List() : n(0), m(0) {}
    int size() { return n + m; }
};
```

这里我们使用了标准库提供的 vector 来简化动态内存管理。

12.4.2 插入和添加

令函数 $\textsc{Front}(L)$ 返回 front 数组，而 $\textsc{Rear}(L)$ 返回 rear 数组。简单起见，假设数组支持动态长度调整。插入和删除可以实现如下。

function INSERT(L, x)
  $F \leftarrow$ FRONT(L)
  SIZE(F) $\leftarrow$ SIZE(F) + 1
  F[SIZE(F)] $\leftarrow x$

function APPEND(L, x)
  $R \leftarrow$ REAR(L)
  SIZE(R) $\leftarrow$ SIZE(R) + 1
  R[SIZE(R)] $\leftarrow x$

由于所有的操作都是在 front 数组和 rear 数组的末尾进行的，它们都是常数时间 $O(1)$ 的。下面的 ISO C++ 例子程序实现了这一算法。

```
template<typename Key>
void insert(List<Key>& xs, Key x) {
    ++xs.n;
    xs.front.push_back(x);
}

template<typename Key>
void append(List<Key>& xs, Key x) {
    ++xs.m;
    xs.rear.push_back(x);
}
```

12.4.3 随机访问

由于内部的数据结构是数组（动态数组 vector），它支持随机访问，因此很容易实现索引算法。

function GET(L, i)
  $F \leftarrow$ FRONT(L)
  $n \leftarrow$ SIZE(F)
  if $i \leq n$ **then**
    return $F[n - i + 1]$
  else
    return REAR(L)[$i - n$]

这里索引 $i \in [1, |L|]$。如果它不大于 front 数组的大小，则待访问的元素存储于 front 数组中。但因为 front 数组和 rear 数组是按照头对头的方式连接到一起的，所以 front 数组中的元素是按照“逆序”索引的。我们需要用数组的尺寸减去 i 来定位到元素；如果 i 大于 front 数组的大小，说明待访问的元素在 rear 数组中。而 rear 数组中的各个元素的顺序是正常的，我们只需要将 i 减去 front 数组的尺寸就可以定位到元素。

下面的 ISO C++ 例子程序实现了这一算法。

```
template<typename Key>
Key get(List<Key>& xs, int i) {
    if( i < xs.n )
        return xs.front[xs.n-i-1];
    else
        return xs.rear[i-xs.n];
}
```

随机访问修改元素的算法留给读者作为练习。

12.4.4 删除和平衡

删除要比插入和添加复杂。这是因为删除可能造成一个数组（front 数组或者 rear 数组）变空，而另外一个仍存有元素。极端情况下，列表会变得很不平衡。我们需要在这种情况下恢复平衡。

最简单的想法是当 front 或者 rear 数组变空时开始修复。我们可以将另外一个数组分成两半，然后将前一半反转顺序形成一对新的数组。算法描述如下：

function BALANCE(L)
  $F \leftarrow \text{FRONT}(L)$, $R \leftarrow \text{REAR}(L)$
  $n \leftarrow \text{SIZE}(F)$, $m \leftarrow \text{SIZE}(R)$
  if $F = \phi$ **then**
    $F \leftarrow \text{REVERSE}(R[1 \ldots \lfloor \frac{m}{2} \rfloor])$
    $R \leftarrow R[\lfloor \frac{m}{2} \rfloor + 1 \ldots m]$
  else if $R = \phi$ **then**
    $R \leftarrow \text{REVERSE}(F[1 \ldots \lfloor \frac{n}{2} \rfloor])$
    $F \leftarrow F[\lfloor \frac{n}{2} \rfloor + 1 \ldots n]$

实际上 front 数组变空或 rear 数组变空引发的恢复平衡操作是对称的。我们可以交换 front 和 rear 数组，递归调用平衡恢复函数，最后在再把 front 和 rear 数组交换回来。下面的 ISO C++ 例子程序实现了这一思路。

```
template<typename Key>
void balance(List<Key>& xs) {
    if(xs.n == 0) {
        back_insert_iterator<vector<Key> > i(xs.front);
        reverse_copy(xs.rear.begin(), xs.rear.begin() + xs.m/2, i);
        xs.rear.erase(xs.rear.begin(), xs.rear.begin() +xs.m/2);
        xs.n = xs.m/2;
        xs.m -= xs.n;
    } else if(xs.m == 0) {
        swap(xs.front, xs.rear);
        swap(xs.n, xs.m);
        balance(xs);
        swap(xs.front, xs.rear);
        swap(xs.n, xs.m);
```

```
    }
}
```

定义好 BALANCE 算法后，就很容易实现头部和尾部的删除算法了。

function REMOVE-HEAD(L)
    BALANCE(L)
    $F \leftarrow$ FRONT(L)
    if $F = \phi$ **then**
        REMOVE-TAIL(L)
    else
        SIZE(F) $\leftarrow$ SIZE(F) - 1

function REMOVE-TAIL(L)
    BALANCE(L)
    $R \leftarrow$ REAR(L)
    if $R = \phi$ **then**
        REMOVE-HEAD(L)
    else
        SIZE(R) $\leftarrow$ SIZE(R) - 1

这里存在一种边界情况：恢复平衡后，待删除的数组仍然是空的。此时，使用双数组实现的列表中只有一个元素。我们只需要删除掉这唯一的元素，结果是一个空列表。下面的 ISO C++ 程序实现了这一算法。

```
template<typename Key>
void remove_head(List<Key>& xs) {
    balance(xs);
    if(xs.front.empty())
        remove_tail(xs); //删除rear中的唯一元素。
    else {
        xs.front.pop_back();
        --xs.n;
    }
}

template<typename Key>
void remove_tail(List<Key>& xs) {
    balance(xs);
    if(xs.rear.empty())
        remove_head(xs); //删除front中的唯一元素。
    else {
        xs.rear.pop_back();
        --xs.m;
    }
}
```

显然，最坏情况下性能为 $O(n)$，其中 n 是双数组列表中存储的元素个数。最坏情况发生在平衡恢复被启动时，不论是逆序操作，还是 shift 操作都是线性时间的。但是删除算法的分摊复杂度仍然是 $O(1)$，我们把证明作为练习留给读者。

练习 12.3

1. 选择一门命令式语言实现随机访问的修改算法。
2. 我们使用了标准库中的 vector 来处理动态内存分配，请尝试用普通数组，自己管理内存分配来实现双数组列表，比较两个版本并分析算法的复杂度是否受到了影响。
3. 证明双数组列表删除算法的分摊复杂度为 $O(1)$。

12.5 可连接列表

通过使用二叉随机访问列表，我们实现了序列数据结构。支持在 $O(\lg n)$ 时间的头部插入和删除，以及通过索引进行随机存取。

但是，将两个列表连接起来却并不容易。每个列表都是由完全二叉树组成的森林，我们不能简单地将它们合并到一起（森林本质上是树的列表，对于任何 size，最多只存在一棵树的尺寸等于这一 size。而且直接合并两个列表也并不快）。一个方法是将第一个序列中的元素逐一推入一个栈中，然后依次将栈中元素弹出并使用 `cons` 函数插入到第二个序列中。当然，栈可以通过使用递归来隐式实现，例如：

$$concat(s_1, s_2) = \begin{cases} s_2 & : & s_1 = \phi \\ cons(head(s_1), concat(tail(s_1), s_2)) & : & otherwise \end{cases} \tag{12.16}$$

其中函数 $cons$、$head$ 和 $tail$ 的定义和此前的一致。

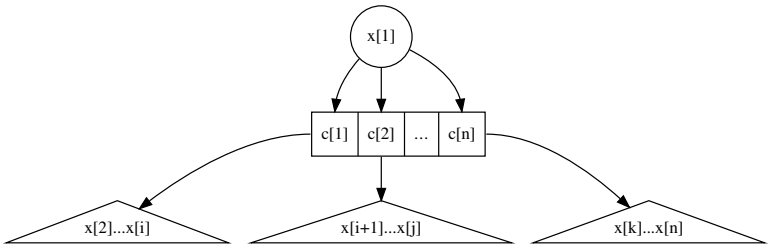
如果两个序列的长度分别是 m 和 n，这一方法首先用 $O(n \lg n)$ 时间将第一个序列中的元素推入栈中，然后使用 $\Omega(n \lg(n+m))$ 的时间将元素逐一插入的第二个序列的前面。其中 Ω 表示上限，具体的定义可以参考 [4]。

在前面章节中，我们实现了实时队列。他支持常数时间 $O(1)$ 的入队和出队。如果我们能够将队列的连接操作实现为某种形式的入队操作，就可以将性能提高到常数时间。Okasaki 在 [3] 中给出了这样的实现。

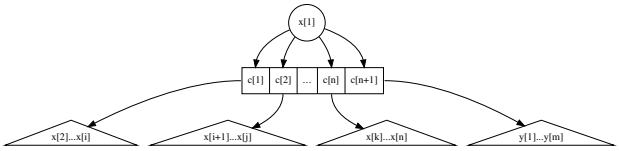
为了实现可连接列表，Okasaki 设计了一种 K 叉树结构。树的根节点保存列表中的第一个元素，我们可以用常数时间 $O(1)$ 访问到它。所有子树都是更小的可连接列表，保存在一个实时队列中。将另外一个列表连接到尾部相当于把这个列表添加为最后一个子树，这实际上是一个入队操作。添加新元素可以这样实现：首先将元素放入到一棵只有一个叶子节点的树中。然后将这棵树连接到序列的尾部从而完成添加。

图12.6描述了这一数据结构。

下面的 Haskell 例子程序定义了这种递归数据结构。



(a) 列表 $\{x_1, x_2, ..., x_n\}$ 的数据结构



(b) 和另一列表 $\{y_1, y_2, ..., y_m\}$ 连接后的结果

图 12.6: 可连接列表的数据结构

```
data CList a = Empty | CList a (Queue (CList a))
```

一个可连接列表或者为空，或者是一个 K 叉树，包含一个 root 元素和一个存有 K 棵子树的队列，每棵子树也都是可连接列表。这里我们复用前面章节中定义的实时队列。

设函数 $clist(x, Q)$ 通过一个元素 x 和一个由子列表组成的队列 Q 构造一个可连接列表。函数 $root(s)$ 返回 K 叉树的根元素，$queue(s)$ 返回由子序列构成的队列。定义连接算法可以定义如下：

$$concat(s_1, s_2) = \begin{cases} s_1 & : & s_2 = \phi \\ s_2 & : & s_1 = \phi \\ clist(x, push(Q, s_2)) & : & otherwise \end{cases} \tag{12.17}$$

其中 $x = root(s_1)$、$Q = queue(s_1)$。连接两个列表时，如果任何一个为空，则结果为另一个列表；否则，我们将第二个列表放入第一个列表中队列的尾部。

使用实时队列可以保证入队操作的性能为常数时间 $O(1)$，因此列表连接操作的时间为 $O(1)$。

下面的 Haskell 例子程序实现了连接算法。

```
concat x Empty = x
concat Empty y = y
concat (CList x q) y = CList x (push q y)
```

这一设计既获得了良好的连接操作性能，还同时可以在头部、尾部高效添加元素

$$cons(x, s) = concat(clist(x, \phi), s) \tag{12.18}$$

$$append(s, x) = concat(s, clist(x, \phi)) \tag{12.19}$$

通过返回 K 叉树的根，可以获得列表的第一个元素

$$head(s) = root(s) \tag{12.20}$$

但是从可连接列表的头部删除元素有些复杂。这是因为第一个元素为根节点，删除后我们需要从剩余的子树队列中重新构造一棵 K 叉树。

根节点被删除后，剩下的所有子树也都是由 K 叉树表示的可连接列表。我们可以把它们全部连接到一起，形成一个新列表。

$$concatAll(Q) = \begin{cases} \phi & : \quad Q = \phi \\ concat(front(Q), concatAll(pop(Q))) & : \quad otherwise \end{cases} \tag{12.21}$$

其中函数 $front$ 返回队列中的第一个元素而不将其删除，而 pop 会执行出队操作。

如果队列为空，说明不存在子树，因此结果为一个空列表；否则，我们将第一棵子树出队，它是一个可连接列表，然后递归地将剩余的子树连接在一起；最后，再把递归连接的结果置于第一棵子树的末尾。

定义好 $concatAll$ 后，我们就可以实现从列表头部删除元素的算法了。

$$tail(s) = concatAll(queue(s)) \tag{12.22}$$

下面的 Haskell 例子程序实现了这一算法。

```
head (CList x _) = x
tail (CList _ q) = concatAll q

concatAll q | isEmptyQ q = Empty
            | otherwise = concat (front q) (concatAll (pop q))
```

函数 `isEmptyQ` 用来判断一个队列是否为空，它的实现很简单，这里不再赘述。读者可以参考本书附带的例子代码。

算法 $concatAll$ 实际上遍历了队列，逐步归并（reduce）到一个最终的结果。这和我们在二叉搜索树一章中介绍的fold 概念非常类似。读者可以参考本书的附录了解 fold 的详细定义和解释。我们可以为队列也定义 fold 操作[2][10]。

$$foldQ(f, e, Q) = \begin{cases} e & : \quad Q = \phi \\ f(front(Q), foldQ(f, e, pop(Q))) & : \quad otherwise \end{cases} \tag{12.23}$$

函数 $foldQ$ 接受三个参数，一个二元函数 f，用以 reduce，一个初始值 e，和一个需要遍历的队列 Q。

我们给出一些在队列上进行 fold 的例子。假设队列 Q 从头到尾包含元素 $\{1, 2, 3, 4, 5\}$。

[2]某些函数式编程语言，例如 Haskell，定义了概念为 monoid（幺半群）的 type class，可以方便地在自定义的数据结构上进行 fold。

$$foldQ(+, 0, Q) = 1 + (2 + (3 + (4 + (5 + 0)))) = 15$$
$$foldQ(\times, 1, Q) = 1 \times (2 \times (3 \times (4 \times (5 \times 1)))) = 120$$
$$foldQ(\times, 0, Q) = 1 \times (2 \times (3 \times (4 \times (5 \times 0)))) = 0$$

仿照这些例子，函数 $concatAll$ 可以用 $foldQ$ 来定义。

$$concatAll(Q) = foldQ(concat, \phi, Q) \tag{12.24}$$

下面的 Haskell 例子程序使用 fold 的概念实现了子树的连接。

```
concatAll = foldQ concat Empty

foldQ :: (a → b → b) → b → Queue a → b
foldQ f z q | isEmptyQ q = z
            | otherwise = (front q) `f` foldQ f z (pop q)
```

但是删除算法的性能并非在所有的情况下都能得到保证。最坏情况发生在用户连续向一个空列表添加 n 个元素后，然后立即执行一次删除时。此时 K 叉树中，第一个元素存储在根节点，而 $n-1$ 棵子树都只含有一个叶子节点。因此 $concatAll$ 算法退化到线性时间 $O(n)$。

元素的插入、添加、删除、以及列表的连接操作如果随机发生，平均下来，这一算法的分摊复杂度是常数时间 $O(1)$ 的。具体证明我们留给读者作为练习。

练习 12.4

1. 如何将一个元素添加到二叉随机访问列表的末尾?
2. 证明可连接列表的删除操作的分摊复杂度为常数时间 $O(1)$，提示：使用 banker 方法。
3. 选择一门命令式语言，实现可连接列表。

12.6 手指树（Finger Tree）

我们目前介绍的这些数据结构，尚未达到本章开头给出的全部目标。

二叉随机访问列表可以在头部进行快速地插入、删除，随机访问的速度也比较快。但是两个列表连接的性能不够理想，同时也没有好方法在尾部添加元素。

可连接列表可以快速地将多个列表连接起来，在头部和尾部插入的性能也很好。但是却不支持通过索引随机访问元素。

这两个例子提供了一些思路给我们:

- 为了能够在头部和尾部进行快速操作，必须能够通过某种方式快速地访问头部和尾部;
- 类似于树的数据结构可以帮助将随机访问转换成分而治之的搜索，如果树的平衡性很好，则搜索性能可以保证为对数时间。

12.6.1 定义

手指树（finger tree）[66] 于 1977 年被提出，可以用于实现高效的序列。并且适于纯函数式的实现 [65]。

我们知道，树的平衡与否，对于搜索的性能至关重要。因此可以考虑使用平衡树作为底层数据结构。手指树的底层数据结构为 2-3 树，它是一种特殊的 B-树（读者可以参考本书前面 B 树的章节）。

一棵 2-3 树包含两个或三个子节点。下面的 Haskell 例子程序定义了 2-3 树。

```
data Node a = Br2 a a | Br3 a a a
```

在命令式编程环境中，节点的定义可以包含一个子节点的列表，这一列表至多包含 3 个子节点。下面的 ANSI C 例子程序定义了这样的节点。

```
union Node {
    Key* keys;
    union Node* children;
};
```

这一定义中，一个节点可以包含 $2 \sim 3$ 个 key 或者 $2 \sim 3$ 棵子树。这里 key 是叶子节点中元素的类型。

记最左侧的非叶子节点为 front 手指（也称左侧手指），最右侧的非叶子节点为 rear 手指（也称右侧手指）。两个手指都是 2-3 树，并且所有的子节点都是叶子，我们可以直接用含有 2 到 3 个叶子的列表表示它们。当然手指树也可以为空，或者是仅含有一个元素的叶子。

归纳起来，一棵手指树可以被描述如下：

- 一棵手指树或者为空；
- 或者是仅包含一个元素的叶子；
- 或者包含三部分：一个左侧手指，是一个至多包含 3 个元素的列表；一棵子手指树；和一个右侧手指，它也是一个至多包含 3 个元素的列表。

这一定义是递归的，可以容易地用纯函数式的方式实现。下面的 Haskell 例子程序定义了手指树。

```
data Tree a = Empty
            | Lf a
            | Tr [a] (Tree (Node a)) [a]
```

在命令式环境中，我们可以用类似的方法定义手指树。而且，我们还可以增加一个指向 parent 的变量，方便从任何节点回溯到根节点。下面的 ANSI C 代码定义了手指树。

```
struct Tree {
    union Node* front;
```

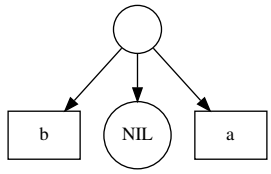
```
    union Node* rear;
    Tree* mid;
    Tree* parent;
};
```

我们可以使用空指针 NIL 代表空树；如果只有一个元素，则将这一元素存储在 front 手指中。而它的 rear 手指和中间部分都为空。

图12.7和12.8给出了一些手指树的例子

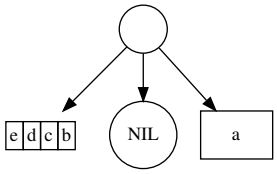
(a) 空树

(b) 只含有一个叶子

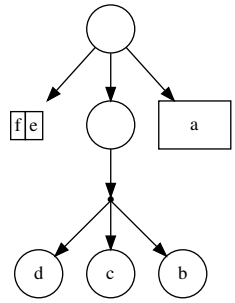


(c) front 和 rear 手指都只含有一个元素，中间部分为空。

图 12.7: 手指树的例子，1



(a) 向 front 手指插入 3 个元素后，它超过了 2-3 树的限制，不再平衡。



(b) 恢复平衡。front 手指含有 2 个元素；中间部分为一个叶子，包含一棵有 3 个分支的 2-3 树。

图 12.8: 手指树的例子，2

第一个例子为一棵空树；第二个例子是插入一个元素后的结果，变成了只含有一个叶子的节点；第三个例子是含有两个元素的手指树，一个元素在 front 手指中，另外一个在 rear 手指中。

如果我们继续向树中插入元素，这些新元素将依次放入 front 手指中，直到超过 2-3 树的限制。第四个例子给出了这种情况，front 手指中保存了 4 个元素，树不再平衡。

最后的例子中，树恢复了平衡。front 手指中存有 2 个元素。中间部分不再为空，而是一个含有 2-3 树的“叶子”（我们稍后解释为什么它是叶子）。叶子的内容为一棵有 3 个分支的树，每个分支都包含了一个元素。

下面的 Haskell 表达式对应这 5 个例子。

```
Empty
Lf a
[b] Empty [a]
[e, d, c, b] Empty [a]
[f, e] Lf (Br3 d c b) [a]
```

在最后一个例子中，为什么中间部分的子树是一个叶子呢？手指树的定义是递归的。除去 front 和 rear 手指的中间部分是一棵更深的手指树，定义为 $Tree(Node(a))$。每当深度增加时，$Node$ 就会被多嵌入一级。如果第一级的元素类型为 a，则第二级的元素类型为 $Node(a)$，第三级为 $Node(Node(a))$……第 n 级为 $Node(Node(Node(...(a))...)) = Node^n(a)$，其中 ^{n} 表示 $Node$ 函数被施行（apply）了 n 次。

12.6.2 向序列的头部插入元素

我们给出的例子实际描述了向手指树逐一插入元素的典型过程。归纳这些例子可以给出在头部插入的算法描述。

当向一棵手指树 T 中插入元素 x 时，

- 如果树为空，则结果为一个叶子，包含一个元素 x；
- 如果树为一个叶子，包含元素 y，则结果为一棵新手指树。front 手指包含了新插入的元素 x，rear 手指包含了此前的元素 y；中间部分为空的手指树；
- 如果 front 手指中包含的元素个数不超过 2-3 树的上限 3，新元素被插入到 front 手指的最前面；
- 否则，如果 front 手指中包含的元素个数超过了 2-3 树的上限。front 手指中的最后 3 个元素被放入一棵新的 2-3 树，然后这棵树被递归地插入到中间部分。新元素 x 被插入到 front 手指中剩余元素的前面。

令函数 $leaf(x)$ 创建包含元素 x 的叶子节点，函数 $tree(F, T', R)$ 从三部分构造出一棵手指树：F 为 front 手指，是一个包含若干元素的列表。R 是 rear 手指；T' 是中间部分的子树，为一棵更深的手指树。函数 $tr3(a, b, c)$ 从三个元素 a, b, c 创建一

棵 2-3 树；函数 $tr2(a, b)$ 从两个元素 a 和 b 创建一棵 2-3 树。

$$insertT(x, T) = \begin{cases} leaf(x) & : & T = \phi \\ tree(\{x\}, \phi, \{y\}) & : & T = leaf(y) \\ tree(\{x, x_1\}, insertT(& : & T = tree(\{x_1, x_2, x_3, x_4\}, \\ \quad tr3(x_2, x_3, x_4), T'), R) & & \quad T', R) \\ tree(\{x\} \cup F, T', R) & : & otherwise \end{cases} \tag{12.25}$$

这一算法的性能主要由递归部分决定。其他情况下都是常数时间 $O(1)$。递归的深度和树的高度成比例，因此算法的性能为 $O(h)$，其中 h 为树的高度。由于使用 2-3 树并维持平衡，因此 $h = O(\lg n)$，其中 n 是手指树中存储元素的个数。

更深入的分析可以给出 $insertT$ 的分摊复杂度为 $O(1)$，递归情况消耗的时间可以分摊到其他简单情况中。读者可以参考 [3] 和 [65] 来了解详细的证明。

下面的 Haskell 例子程序实现了插入算法。

```
cons :: a → Tree a → Tree a
cons a Empty = Lf a
cons a (Lf b) = Tr [a] Empty [b]
cons a (Tr [b, c, d, e] m r) = Tr [a, b] (cons (Br3 c d e) m) r
cons a (Tr f m r) = Tr (a:f) m r
```

这段程序使用了 LISP 的命名传统来描述在如何列表的前面插入元素。

插入算法也可以用命令式的方式实现。设函数 TREE() 可以构造一棵空树，树中的所有变量包括 front 手指、rear 手指、中间部分子树、和父节点指针都为空。函数 NODE() 用以创建一个空节点。

function PREPEND-NODE(n, T)
  $r \leftarrow$ TREE()
  $p \leftarrow r$
  CONNECT-MID(p, T)
  while FULL?(FRONT(T)) **do**
    $F \leftarrow$ FRONT(T) ▷ $F = \{n_1, n_2, n_3, ...\}$
    FRONT(T) $\leftarrow \{n, F[1]\}$ ▷ $F[1] = n_1$
    $n \leftarrow$ NODE()
    CHILDREN(n) $\leftarrow F[2..]$ ▷ $F[2..] = \{n_2, n_3, ...\}$
    $p \leftarrow T$
    $T \leftarrow$ MID(T)
  if $T =$ NIL **then**
    $T \leftarrow$ TREE()
    FRONT(T)$\leftarrow \{n\}$
  else if $|$ FRONT(T) $| = 1 \wedge$ REAR(T) $= \phi$ **then**
    REAR(T) $\leftarrow$ FRONT(T)

```
        FRONT(T) ← {n}
    else
        FRONT(T) ← {n}∪ FRONT(T)
    CONNECT-MID(p, T) ← T
    return FLAT(r)
```

其中符号 $L[i..]$ 表示列表 L 的子列表，不包含最前面的 $i-1$ 个元素，若 $L = \{a_1, a_2, ..., a_n\}$, 则 $L[i..] = \{a_i, a_{i+1}, ..., a_n\}$。

函数 FRONT、REAR、MID、和 PARENT 分别用以获得 front 手指、rear 手指、中间部分子树、和父节点。函数 CHILDREN 用以获取一个节点的全部子节点。

函数 CONNECT-MID(T_1, T_2) 将树 T_2 作为中间部分子树连接到 T_1 上。如果 T_2 不为空，还会将 T_1 设置为 T_2 的父节点。

在这一算法中，如果 front 手指已满，达到 3 个元素，我们就沿着中间部分子树，进行一次性的自顶向下的遍历。我们将 front 手指中除第一个元素外的剩余部分抽出，放入一个新节点（$Node$ 深度增加 1），然后继续将此新节点插入到中间部分的子树中。front 手指剩下原来的第一个元素，待插入元素被放到最前面成为新的第一个元素。

遍历结束后，我们要么到达了一个空树，要么到达了一棵子树，这棵子树的 front 手指仍然可以容纳更多元素。对于第一种情况，我们创建一个叶子节点，对于后一种情况，我们进行一次简单的列表插入将节点插入到 front 手指的最前面。

在遍历过程中，我们使用 p 来记录当前树的父节点，这样新创建的树可以被连接到 p 的中间部分作为子树。

最后，我们返回树的根 r。算法中最后需要解释的是 FLAT 函数。为了简化逻辑，我们创建了一棵空的“ground”树，并让它成为根的父节点。在返回根节点前，我们需要再消除掉这棵多余的“ground”树。这一过程的实现如下：

```
function FLAT(T)
    while T ≠ NIL ∧T is empty do
        T ← MID(T)
    if T ≠ NIL then
        PARENT(T) ← NIL
    return T
```

这个 while 循环用以检查树 T 是否是 ground，即树不为 NIL（$= \phi$），但是 front 手指和 rear 手指都是空。

下面的 Python 例子程序实现了手指树的插入算法。

```
def insert(x, t):
    return prepend_node(wrap(x), t)

def prepend_node(n, t):
    root = prev = Tree()
```

```
    prev.set_mid(t)
    while frontFull(t):
        f = t.front
        t.front = [n] + f[:1]
        n = wraps(f[1:])
        prev = t
        t = t.mid
    if t is None:
        t = leaf(n)
    elif len(t.front)==1 and t.rear == []:
        t = Tree([n], None, t.front)
    else:
        t = Tree([n]+t.front, t.mid, t.rear)
    prev.set_mid(t)
    return flat(root)

def flat(t):
    while t is not None and t.empty():
        t = t.mid
    if t is not None:
        t.parent = None
    return t
```

函数 set_mid、frontFull、wrap、wraps、empty、和树的创建的实现都很简单直观，我们这里不再赘述，读者可以把它们当作练习。

12.6.3 从头部删除元素

通过把 $insertT$ 中的各个操作进行逆操作，就可以实现从序列头部删除元素。

记 $F = \{f_1, f_2, ...\}$ 为 front 手指列表，M 为中间部分子树，$R = \{r_1, r_2, ...\}$ 为 rear 手指列表，$R' = \{r_2, r_3, ...\}$ 为 R 中除去第一个元素外的剩余部分。

$$extractT(T) = \begin{cases} (x, \phi) & : & T = leaf(x) \\ (x, leaf(y)) & : & T = tree(\{x\}, \phi, \{y\}) \\ (x, tree(\{r_1\}, \phi, R')) & : & T = tree(\{x\}, \phi, R) \\ (x, tree(toList(F'), M', R)) & : & \begin{array}{l} T = tree(\{x\}, M, R), \\ (F', M') = extractT(M) \end{array} \\ (f_1, tree(\{f_2, f_3, ...\}, M, R)) & : & otherwise \end{cases} \tag{12.26}$$

其中函数 $toList(T)$ 将一棵 2-3 树转换为一个普通列表。如下：

$$toList(T) = \begin{cases} \{x, y\} & : & T = tr2(x, y) \\ \{x, y, z\} & : & T = tr3(x, y, z) \end{cases} \tag{12.27}$$

我们略过了错误处理（例如从一个空树中删除元素的错误）。如果手指树是只包含一个元素的叶子，则删除的结果为一棵空树；如果手指树包含两个元素，一个元素在 front 手指，另一个元素在 rear 手指，我们删除 front 手指中的元素，结果变成了

只含有一个元素的叶子；如果 front 手指中只含有一个元素，中间部分为空，而 rear 手指不空，我们删除 front 手指中的唯一元素，然后从 rear 手指中“借”一个元素放入 front 手指；如果 front 手指只有一个元素，而中间部分的子树不为空，我们就递归地从子树中删除一个节点，然后将这一节点转换成普通列表来代替 front 手指。而原来 front 手指中的唯一元素被删除并返回；最后一种情况，如果 front 手指包含一个以上的元素，我们只需要将第一个元素删除，而维持其它部分不变。

图12.9展示了从序列头部删除两个元素的例子。手指树中存有 10 个元素。当第一个元素被删除后，front 手指中还剩一个元素。接着再次删除一个元素后，front 手指变为空。我们从中间的子树中“借”一个节点。把它从一棵 2-3 树转换成含有 3 个元素的列表。这一列表成为新的 front 手指。中间部分的子树从原来的三部分变成一个只含有一个 2-3 树节点的叶子。这一节点包含三个元素。

下面的 Haskell 例子程序实现了 `uncons`。

```
uncons :: Tree a → (a, Tree a)
uncons (Lf a) = (a, Empty)
uncons (Tr [a] Empty [b]) = (a, Lf b)
uncons (Tr [a] Empty (r:rs)) = (a, Tr [r] Empty rs)
uncons (Tr [a] m r) = (a, Tr (nodeToList f) m' r) where (f, m') = uncons m
uncons (Tr f m r) = (head f, Tr (tail f) m r)
```

其中函数 `nodeToList` 定义如下：

```
nodeToList :: Node a → [a]
nodeToList (Br2 a b) = [a, b]
nodeToList (Br3 a b c) = [a, b, c]
```

类似地，我们可以使用 `uncons` 定义函数 `head` 和 `tail`。

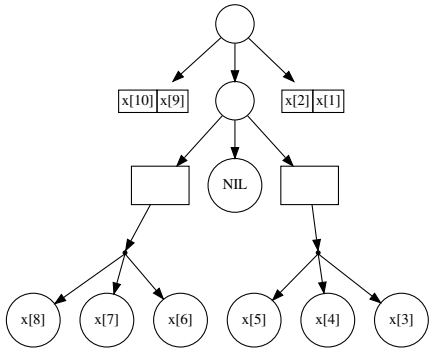
```
head = fst ∘ uncons
tail = snd ∘ uncons
```

12.6.4 删除时处理不规则的手指树

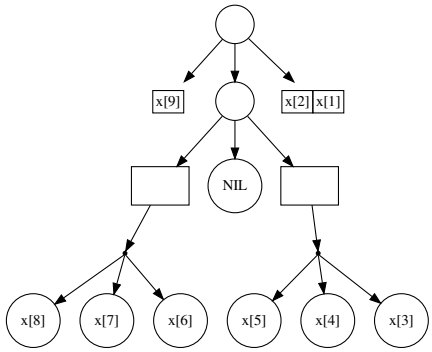
删除的算法可以归纳为“删除——借”策略。如果删除后 front 手指变空，就从中间部分的子树中“借”节点。但是树的形式有可能是不规则的，例如 front 手指和中间子树都为空。这种情况通常是由于分拆操作造成的，我们稍后会详细介绍。

我们要设计一个命令式算法，可以从不规则的手指树中删除第一个元素。思路是首先进行一轮自顶向下的遍历，找到一棵子树，或者它的 front 手指不为空，或者它的 front 手指和中间部分的子树都为空，如图12.10。对于前者，我们可以从 front 手指中提取出第一个元素，它为一个节点。对于后者，由于只有 rear 手指不为空，我们可以把它和空的 front 手指交换，将其转换成前一种情况。

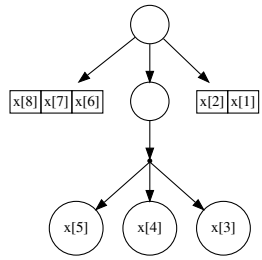
此后，我们需要检查从 front 手指中取出的节点是否为叶子节点（如何做到？我们将这一问题留给读者作为练习），如果不是，我们需要继续从这一节点的子节点中提取出第一个子节点，而把剩余的节点列表作为当前树的父节点的 front 手指。我们



(a) 用手指树表示的含有 10 个元素的序列。



(b) 删除第一个元素后，front 手指还剩一个元素。



(c) 再次从头部删除一个元素，我们从中间部分的子树“借”一个节点，将这个节点从一棵 2-3 树转换成一个列表，作为新的 front 手指。中间部分的子树变成了含有一棵 2-3 树节点的叶子。

图 12.9: 从一个序列的头部依次删除两个元素的例子

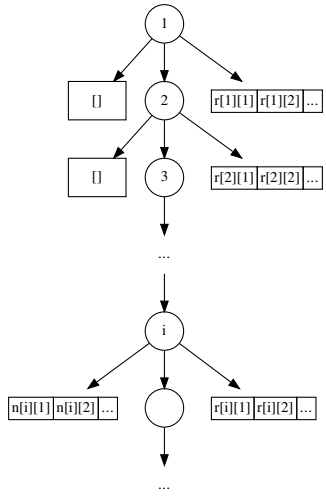


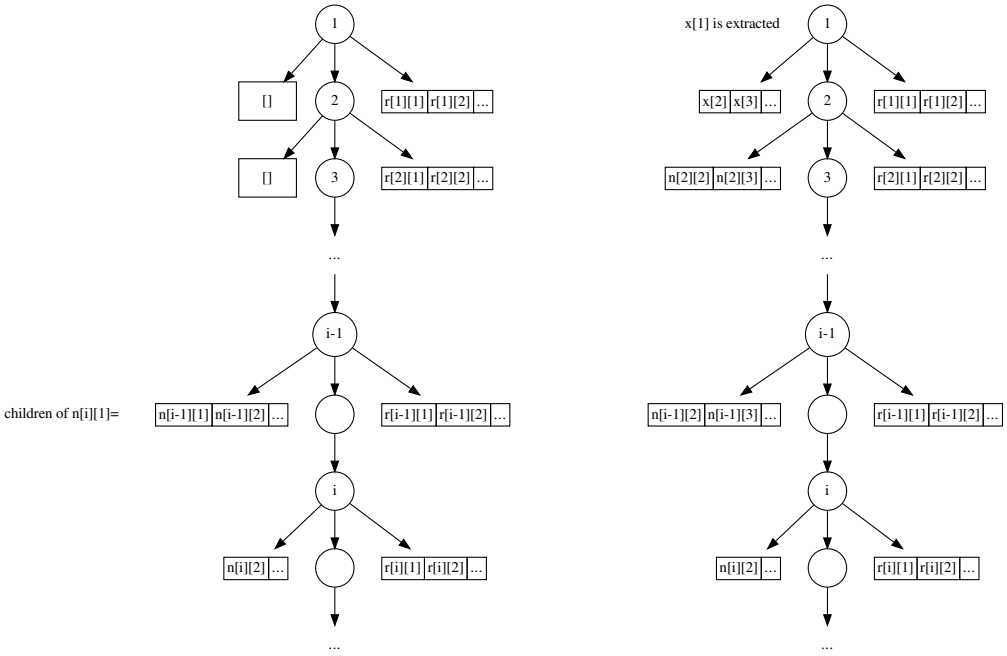
图 12.10: 不规则树的例子，第 i 层子树的 front 手指不为空

需要沿着父节点一直向上回溯，直到我们提取到一个叶子节点。此时我们将到达树的根节点。图12.11描述了这一过程。

根据这一思路，下面的算法实现了列表头部的删除操作。这里假设传入的树不为空。

```
function EXTRACT-HEAD(T)
    r ← TREE()
    CONNECT-MID(r, T)
    while FRONT(T) = φ∧ MID(T) ≠ NIL do
        T ← MID(T)
    if FRONT(T) = φ∧ REAR(T) ≠ φ then
        EXCHANGE FRONT(T) ↔ REAR(T)
    n ← NODE()
    CHILDREN(n) ← FRONT(T)
    repeat
        L ← CHILDREN(n)                          ▷ L = {n1, n2, n3, ...}
        n ← L[1]                                 ▷ n ← n1
        FRONT(T) ← L[2..]                        ▷ L[2..] = {n2, n3, ...}
        T ← PARENT(T)
        if MID(T) becomes empty then
            MID(T) ← NIL
    until n is a leaf
    return (ELEM(n), FLAT(r))
```

这里函数 ELEM(n) 返回叶子节点 n 中保存的唯一元素。和命令式插入算法类似，



(a) 提取第一个元素 n[i][1]，然后将它的子节点放到上一级树的 front 手指中。

(b) 重复这一过程 i 次，最终提取到 x[1]。

图 12.11: 自底向上遍历，直到提取出一个叶子节点

这里使用了一棵“ground”树作为根节点的父节点。这样可以简化边界处理的逻辑。因此最后在返回前要做额外的处理，去除掉多余的“grand”树。

下面的 Python 例子程序实现了这一算法。

```
def extract_head(t):
    root = Tree()
    root.set_mid(t)
    while t.front == [] and t.mid is not None:
        t = t.mid
    if t.front == [] and t.rear ≠ []:
        (t.front, t.rear) = (t.rear, t.front)
    n = wraps(t.front)
    while True: #repeat-until 循环
        ns = n.children
        n = ns[0]
        t.front = ns[1:]
        t = t.parent
        if t.mid.empty():
            t.mid.parent = None
            t.mid = None
        if n.leaf:
            break
    return (elem(n), flat(root))
```

如果树的 front 手指和 rear 手指都为空，则成员函数 `Tree.empty()` 返回真。我们增加了一个标记 `Node.leaf` 来记录一个节点是叶子节点，还是复合节点。本节的练习要求读者思考其他的方法。

由于允许不规则的树存在，从手指树中获取第一个和最后一个元素的算法需要做相应的调整，对于不规则的树，手指可以为空，我们不再仅仅返回手指的第一个或者最后一个子节点。

解决方法和 Extract-Head 类似，如果手指为空，而中间部分的子树不为空，我们就沿着中间部分向下遍历，直到发现手指不为空，或者所有的节点都存储在另一侧的手指中。例如下面的算法，即使是不规则树，也能返回第一个叶子节点。

function First-Lf(T)
  while Front(T) $= \phi \wedge$ Mid(T) $\neq$ NIL **do**
    $T \leftarrow$ Mid(T)
  if Front(T) $= \phi \wedge$ Rear(T) $\neq \phi$ **then**
    $n \leftarrow$ Rear(T)[1]
  else
    $n \leftarrow$ Front(T)[1]
  while n is NOT leaf **do**
    $n \leftarrow$ Children(n)[1]
  return n

注意其中的第二个循环，如果当前的节点不是叶子，它就不断沿着第一个子节点遍历。最后我们会得到一个叶子节点，从而可以获取到元素。

function First(T)
  return Elem(First-Lf(T))

下面的 Python 例子程序实现了这一算法。

```
def first(t):
    return elem(first_leaf(t))

def first_leaf(t):
    while t.front == [] and t.mid is not None:
        t = t.mid
    if t.front == [] and t.rear ≠ []:
        n = t.rear[0]
    else:
        n = t.front[0]
    while not n.leaf:
        n = n.children[0]
    return n
```

获取最后一个元素与此类似，我们将其作为练习留给读者。

12.6.5 在序列的尾部添加元素

由于手指树是对称的，我们可以参考 $insertT$ 实现尾部添加算法。

$$appendT(T,x)=\begin{cases} leaf(x) & : & T=\phi \\ tree(\{y\},\phi,\{x\}) & : & T=leaf(y) \\ tree(F, appendT(M, tr3(x_1,x_2,x_3)), \{x_4,x\}) & : & T=tree(F,M,\{x_1,x_2,x_3,x_4\}) \\ tree(F,M,R\cup\{x\}) & : & otherwise \end{cases} \tag{12.28}$$

如果 rear 手指仍然是合法的 2-3 树，包含的元素不超过 4 个，新元素就可以直接插入到 rear 手指中。否则，我们将 rear 手指分拆，将前三个元素取出，构造一棵新的 2-3 树，递归地添加到中间部分子树的末尾。此外，我们还要处理两种边界情况：一种是手指树为空，另外一种是仅含有一个叶子节点。

下面的 Haskell 例子程序实现了尾部添加算法。

```
snoc :: Tree a → a → Tree a
snoc Empty a = Lf a
snoc (Lf a) b = Tr [a] Empty [b]
snoc (Tr f m [a, b, c, d]) e = Tr f (snoc m (Br3 a b c)) [d, e]
snoc (Tr f m r) a = Tr f m (r++[a])
```

函数的名字 `snoc` 恰好是 `cons` 倒过来，我们以此指出它们操作上的对称关系。

用命令式的方法在尾部添加元素与此类似，下面的算法实现了这一操作。

```
function APPEND-NODE(T, n)
    r ← TREE()
    p ← r
    CONNECT-MID(p, T)
    while FULL?(REAR(T)) do
        R ← REAR(T)                          ▷ R = {n_1, n_2, ..., , n_{m-1}, n_m}
        REAR(T) ← {n, LAST(R) }              ▷ last element n_m
        n ← NODE()
        CHILDREN(n) ← R[1...m − 1]           ▷ {n1, n2, ..., n_{m-1}}
        p ← T
        T ← MID(T)
    if T = NIL then
        T ← TREE()
        FRONT(T) ← {n}
    else if | REAR(T) | = 1 ∧ FRONT(T) = φ then
        FRONT(T) ← REAR(T)
        REAR(T) ← {n}
```

else

$\textsc{Rear}(T) \leftarrow \textsc{Rear}(T) \cup \{n\}$

$\textsc{Connect-Mid}(p, T) \leftarrow T$

return $\textsc{Flat}(r)$

对应的 Python 例子程序如下。

```
def append_node(t, n):
    root = prev = Tree()
    prev.set_mid(t)
    while rearFull(t):
        r = t.rear
        t.rear = r[-1:] + [n]
        n = wraps(r[:-1])
        prev = t
        t = t.mid
    if t is None:
        t = leaf(n)
    elif len(t.rear) == 1 and t.front == []:
        t = Tree(t.rear, None, [n])
    else:
        t = Tree(t.front, t.mid, t.rear + [n])
    prev.set_mid(t)
    return flat(root)
```

12.6.6 从尾部删除元素

和 $appendT$ 类似，我们可以通过实现 $extractT$ 的逆操作从尾部删除最后一个元素。

记非空、非单一叶子的手指树为 $tree(F, M, R)$，其中 F 为 front 手指，M 为中间部分的子树，R 为 rear 手指。

$$removeT(T) = \begin{cases} (\phi, x) & : & T = leaf(x) \\ (leaf(y), x) & : & T = tree(\{y\}, \phi, \{x\}) \\ (tree(init(F), \phi, last(F)), x) & : & T = tree(F, \phi, \{x\}) \wedge F \neq \phi \\ (tree(F, M', toList(R')), x) & : & \begin{array}{l} T = tree(F, M, \{x\}), \\ (M', R') = removeT(M) \end{array} \\ (tree(F, M, init(R)), last(R)) & : & otherwise \end{cases} \tag{12.29}$$

函数 $toList(T)$ 的定义和此前一样，它将一棵 2-3 树转换为普通列表。函数 $init(L)$ 返回列表 L 中除最后一个元素外的剩余部分，若 $L = \{a_1, a_2, ..., a_{n-1}, a_n\}$，则 $init(L) = \{a_1, a_2, ..., a_{n-1}\}$。函数 $last(L)$ 返回列表 L 中的最后一个元素，即 $last(L) = a_n$。读者可以参考本书的附录了解它们的具体实现。

下面的 Haskell 例子程序实现了尾部删除算法。函数被命名为 `unsnoc` 以表明它是函数 `snoc` 的逆运算。

```
unsnoc :: Tree a → (Tree a, a)
unsnoc (Lf a) = (Empty, a)
unsnoc (Tr [a] Empty [b]) = (Lf a, b)
unsnoc (Tr f@(_:_) Empty [a]) = (Tr (init f) Empty [last f], a)
unsnoc (Tr f m [a]) = (Tr f m' (nodeToList r), a) where (m', r) = unsnoc m
unsnoc (Tr f m r) = (Tr f m (init r), last r)
```

我们也可以为手指树定义类似列表的 `last` 和 `init` 函数。

```
last = snd ∘ unsnoc
init = fst ∘ unsnoc
```

命令式的尾部删除算法和头部删除类似。但是这里存在一个特殊情况，当只有一个元素（或者子节点）时，我们总是将其存储在 front 手指，而 rear 手指和中间部分的子树为空（即：$Tree(\{n\}, NIL, \phi)$），如果只从 rear 手指获取最后一个元素，就无法得到正确的结果。

如果 rear 手指为空，这一特殊情况可以通过交换 front 手指和 rear 手指来解决，如下面的算法所示：

function EXTRACT-TAIL(T)
  $r \leftarrow$ TREE()
  CONNECT-MID(r, T)
  while REAR(T) $= \phi \wedge$ MID(T) $\neq$ NIL **do**
    $T \leftarrow$ MID(T)
  if REAR(T) $= \phi \wedge$ FRONT(T) $\neq \phi$ **then**
    EXCHANGE FRONT(T) $\leftrightarrow$ REAR(T)
  $n \leftarrow$ NODE()
  CHILDREN(n) $\leftarrow$ REAR(T)
  repeat
    $L \leftarrow$ CHILDREN(n) ▷ $L = \{n_1, n_2, ..., n_{m-1}, n_m\}$
    $n \leftarrow$ LAST(L) ▷ $n \leftarrow n_m$
    REAR(T) $\leftarrow L[1...m-1]$ ▷ $\{n_1, n_2, ..., n_{m-1}\}$
    $T \leftarrow$ PARENT(T)
    if MID(T) becomes empty **then**
      MID(T) $\leftarrow$ NIL
  until n is a leaf
  return (ELEM(n), FLAT(r))

我们把如何获得最后一个元素，以及这一算法的实现留给读者作为练习。

12.6.7 连接

考虑两棵手指树都不为空的情况。记两棵树为 $T_1 = tree(F_1, M_1, R_1)$ 和 $T_2 = tree(F_2, M_2, R_2)$。可以用 F_1 作为连接结果的新 front 手指，用 R_2 作为连接结果的新 rear 手指。我们需要将 M_1、R_1、F_2、M_2 合并成一棵新的中间部分子树。

由于 R_1 和 F_2 都是节点的列表，所以这一问题等价于实现如下的算法：

$$merge(M_1, R_1 \cup F_2, M_2) = ?$$

进一步观察可以发现，M_1 和 M_2 都是手指树，只不过它们比 T_1 和 T_2 的节点深度大一级，若 T_1 树中存储的元素类型为 a，则 M_1 中存储的元素类型为 $Node(a)$。因此，我们可以递归地进行合并：保留 M_1 的 front 手指和 M_2 的 rear 手指，然后将 M_1 和 M_2 的中间部分，以及 M_1 的 rear 手指和 M_2 的 front 手指合并。

记函数 $front(T)$ 返回树 T 的 front 手指，$rear(T)$ 返回 rear 手指，$mid(T)$ 返回中间部分的子树。当两棵树都不为空时，$merge$ 算法可以定义如下：

$$\begin{aligned} &merge(M_1, R_1 \cup F_2, M_2) = tree(front(M_1), S, rear(M_2)) \\ &S = merge(mid(M_1), rear(M_1) \cup R_1 \cup F_2 \cup front(M_2), mid(M_2)) \end{aligned} \tag{12.30}$$

而树的连接算法也可以使用 $merge()$ 来定义。

$$concat(T_1, T_2) = tree(F_1, merge(M_1, R_1 \cup F_2, M_2), R_2) \tag{12.31}$$

比较这一定义和式（12.30），我们发现连接操作本质上就是合并操作，我们可以给出下面的定义：

$$concat(T_1, T_2) = merge(T_1, \phi, T_2) \tag{12.32}$$

最后，我们需要为 $merge()$ 算法定义边界条件。

$$merge(T_1, S, T_2) = \begin{cases} foldR(insertT, T_2, S) & : & T_1 = \phi \\ foldL(appendT, T_1, S) & : & T_2 = \phi \\ merge(\phi, \{x\} \cup S, T_2) & : & T_1 = leaf(x) \\ merge(T_1, S \cup \{x\}, \phi) & : & T_2 = leaf(x) \\ tree(F_1, merge(M_1, \\ \quad nodes(R_1 \cup S \cup F_2), M2), R_2) & : & otherwise \end{cases} \tag{12.33}$$

大部分的情况都比较直观。若 T_1 或 T_2 中的任何一棵为空，算法就逐一将列表 S 中的元素插入或者添加到另一棵树中；函数 $foldL$ 和 $foldR$ 类似于命令式编程环境中的 for-each 循环。其中 $foldL$ 自左向右处理 S 中的元素，而 $foldR$ 自右向左处理。

对于非空列表 $L = \{a_1, a_2, ..., a_{n-1}, a_n\}$，记 $L' = \{a_2, a_3, ..., a_{n-1}, a_n\}$ 为除第一元素外的剩余部分。$foldL$ 和 $foldR$ 可以分别定义如下：

$$foldL(f, e, L) = \begin{cases} e & : & L = \phi \\ foldL(f, f(e, a_1), L') & : & otherwise \end{cases} \tag{12.34}$$

$$foldR(f, e, L) = \begin{cases} e & : \quad L = \phi \\ f(a_1, foldR(f, e, L')) & : \quad otherwise \end{cases} \tag{12.35}$$

读者可以参考本书的附录了解它们的详细内容。

若任一棵树是仅包含一个元素的叶子，我们将这一元素插入或者添加到 S 中将其转换为前一种边界情况（其中一棵树为空的情况）。

函数 $nodes$ 将元素列表转换成一组 2-3 树的列表。这是因为中间部分的子树中的元素类型，比手指中的元素类型在 $Node$ 上深一级。考虑递归调用达到边界情况的时候，假设此时 M_1 为空，我们需要逐一将所有 $R_1 \cup S \cup F_2$ 中的元素插入 M_2。但是我们不能直接执行插入操作，若此时的元素类型为 a，我们只能将类型为 2-3 树的 $Node(a)$ 插入到 M_2 中。这和前面的 $insertT$ 算法中的处理类似：取出最后三个元素，转换成一棵 2-3 树，然后递归地执行 $insertT$。下面给出了 $nodes$ 的定义：

$$nodes(L) = \begin{cases} \{tr2(x_1, x_2)\} & : \quad L = \{x_1, x_2\} \\ \{tr3(x_1, x_2, x_3)\} & : \quad L = \{x_1, x_2, x_3\} \\ \{tr2(x_1, x_2), tr2(x_3, x_4)\} & : \quad L = \{x_1, x_2, x_3, x_4\} \\ \{tr3(x_1, x_2, x_3)\} \cup nodes(\{x_4, x_5, ...\}) & : \quad otherwise \end{cases} \tag{12.36}$$

函数 $nodes$ 需要遵守 2-3 树的限制条件，若列表中只有 2 个或 3 个元素，结果是只含有一棵 2-3 树的列表；若列表中含有 4 个元素，则结果列表中包含两棵树，每棵树有两个分支；否则，如果多于 4 个元素，就将前 3 个元素放到一棵 2-3 树中，然后递归地调用 $nodes$ 来处理剩余的元素。

连接操作的性能取决于合并算法。分析递归的情况可以发现，递归的深度和两棵树种较矮的一棵成比例。由于 2-3 树可以保证平衡性，它的高度为 $O(\lg n')$ 其中 n' 为元素的个数。合并在边界条件下的性能和插入一样（最多调用 $insertT$ 8 次）为分摊时间 $O(1)$，最坏情况问为 $O(\lg m)$，其中 m 是两棵树的高度差。因此，总体上算法的复杂度为 $O(\lg n)$，其中 n 是两棵手指树中含有的元素总数。

下面的 Haskell 例子程序实现了连接算法

```
concat :: Tree a → Tree a → Tree a
concat t1 t2 = merge t1 [] t2
```

由于 Haskell 标准库 prelude 中含有一个名为 `concat` 的函数，因此需要做一些额外的处理，如隐藏 import 或者更换名字，以避免冲突。

```
merge :: Tree a → [a] → Tree a → Tree a
merge Empty ts t2 = foldr cons t2 ts
merge t1 ts Empty = foldl snoc t1 ts
merge (Lf a) ts t2 = merge Empty (a:ts) t2
merge t1 ts (Lf a) = merge t1 (ts++[a]) Empty
merge (Tr f1 m1 r1) ts (Tr f2 m2 r2) = Tr f1 (merge m1 (nodes (r1 ++ ts ++ f2)) m2) r2
```

其中 $nodes$ 函数的实现如下：

```
nodes :: [a] → [Node a]
nodes [a, b] = [Br2 a b]
nodes [a, b, c] = [Br3 a b c]
nodes [a, b, c, d] = [Br2 a b, Br2 c d]
nodes (a:b:c:xs) = Br3 a b c:nodes xs
```

为了用命令式的方式连接两棵手指树 T_1 和 T_2，我们需要沿着两棵树的中间部分的子树向下遍历，直到其中一棵为空。每次迭代中，我们创建一棵新树，用 T_1 的 front 手指作为新树 T 的 front 手指；用 T_2 的 rear 手指作为 T 的 rear 手指。剩下的两个手指（T_1 的 rear 手指和 T_2 的 front 手指）中的元素被放入一个列表，然后被分组放入若干 2-3 树中。记这一 2-3 树的列表为 N。N 不仅随着遍历在长度上增加，而且每次迭代元素的深度也增加一级。我们将这棵新树附加到上一级的树中作为中间部分，然后进入下一次迭代。

当两棵树种的任一棵树变为空的时候，我们停止遍历，然后逐一将 N 中的 2-3 树插入到另一棵非空的树中，并将其设为上一级结果的中间部分子树。

下面的算法给出了这一过程的详细描述。

function CONCAT(T_1, T_2)
  return MERGE(T_1, ϕ, T_2)

function MERGE(T_1, N, T_2)
  $r \leftarrow$ TREE()
  $p \leftarrow r$
  while $T_1 \neq$ NIL $\wedge T_2 \neq$ NIL **do**
    $T \leftarrow$ TREE()
    FRONT(T) $\leftarrow$ FRONT(T_1)
    REAR(T) $\leftarrow$ REAR(T_2)
    CONNECT-MID(p, T)
    $p \leftarrow T$
    $N \leftarrow$ NODES(REAR(T_1) $\cup N \cup$ FRONT(T_2))
    $T_1 \leftarrow$ MID(T_1)
    $T_2 \leftarrow$ MID(T_2)
  if $T_1 =$ NIL **then**
    $T \leftarrow T_2$
    for each $n \in$ REVERSE(N) **do**
      $T \leftarrow$ PREPEND-NODE(n, T)
  else if $T_2 =$ NIL **then**
    $T \leftarrow T_1$
    for each $n \in N$ **do**
      $T \leftarrow$ APPEND-NODE(T, n)

CONNECT-MID(p, T)
return FLAT(r)

算法中的 for-each 循环也可以用左侧 fold 或者右侧 fold 来实现。下面的 Python 例子程序实现了这一算法。

```
def concat(t1, t2):
    return merge(t1, [], t2)

def merge(t1, ns, t2):
    root = prev = Tree() #作为哨兵的 dummy 节点
    while t1 is not None and t2 is not None:
        t = Tree(t1.size + t2.size + sizeNs(ns), t1.front, None, t2.rear)
        prev.set_mid(t)
        prev = t
        ns = nodes(t1.rear + ns + t2.front)
        t1 = t1.mid
        t2 = t2.mid
    if t1 is None:
        prev.set_mid(foldR(prepend_node, ns, t2))
    elif t2 is None:
        prev.set_mid(reduce(append_node, ns, t1))
    return flat(root)
```

由于 Python 标准库只提供了左侧 fold 的函数 `reduce`，右侧 fold 可以按照下面的定义实现，逐一将元素按照逆序取出并应用传入的函数。

```
def foldR(f, xs, z):
    for x in reversed(xs):
        z = f(x, z)
    return z
```

唯一需要实现的算法是将若干元素平衡分组放入一些 2-3 树中。一棵 2-3 树最多可以容纳 3 个分支，若元素个数多于 4，我们可以取出 3 个放入一棵树中，然后继续处理剩下的元素。若只含有 4 个，则它们被分成 2 棵 2 个分支的树。对于其余的情况（3 个、2 个或 1 个），我们将它们全部放入一棵 2-3 树中。

记节点列表为 $L = \{n_1, n_2, ...\}$，下面的算法实现了这一处理过程。

function NODES(L)
  $N = \phi$
  while $|L| > 4$ **do**
    $n \leftarrow$ NODE()
    CHILDREN$(n) \leftarrow L[1..3]$ ▷ $\{n_1, n_2, n_3\}$
    $N \leftarrow N \cup \{n\}$
    $L \leftarrow L[4...]$ ▷ $\{n_4, n_5, ...\}$
  if $|L| = 4$ **then**
    $x \leftarrow$ NODE()

$\textsc{Children}(x) \leftarrow \{L[1], L[2]\}$
$y \leftarrow \textsc{Node}()$
$\textsc{Children}(y) \leftarrow \{L[3], L[4]\}$
$N \leftarrow N \cup \{x, y\}$
else if $L \neq \phi$ **then**
$n \leftarrow \textsc{Node}()$
$\textsc{Children}(n) \leftarrow L$
$N \leftarrow N \cup \{n\}$
return N

下面的 Python 例子程序实现了这一算法，其中函数 `wraps()` 首先创建一棵树，然后将一个列表中的元素设为节点的子树。

```
def nodes(xs):
    res = []
    while len(xs) > 4:
        res.append(wraps(xs[:3]))
        xs = xs[3:]
    if len(xs) == 4:
        res.append(wraps(xs[:2]))
        res.append(wraps(xs[2:]))
    elif xs ≠ []:
        res.append(wraps(xs))
    return res
```

练习 12.5

1. 选择一门命令式语言，实现完整的手指树插入算法。

2. 如何判定一个节点是否是叶子？它仅包含一个基本元素还是包含一个含有若干子树的复合节点？我们不能仅仅通过 size 来进行判定，例如只包含一个叶子的节点，形如 $node(1, \{node(1, \{x\})\})$。请分别使用动态类型语言（如 Python 或 lisp）和静态类型语言（如 C++）来解决这一问题。

3. 选择一门命令式语言，实现 Extract-Tail 算法。

4. 分别用函数式和命令式的方法返回一棵手指树中的最后一个元素，对于命令式方法，要求能够处理不规则树。

5. 不使用 fold，实现手指树的连接算法。可以使用递归或者循环。

12.6.8 手指树的随机访问

增加 size 记录

提供快速随机访问的策略是将其转换为树搜索。为了避免反复计算树的 size，我们给树和节点增加 size 变量。

下面的 Haskell 例子代码在定义中增加了 size 信息。

```
data Tree a = Empty
            | Lf a
            | Tr Int [a] (Tree (Node a)) [a]
```

下面的 ANSI C 结构定义中也增加了 size 信息。

```
struct Tree {
    union Node* front;
    union Node* rear;
    Tree* mid;
    Tree* parent;
    int size;
};
```

设函数 $tree(s, F, M, R)$ 从 size 信息 s、front 手指列表 F、rear 手指列表 R、和中间部分的子树 M 构造一棵手指树。当我们需要获得 size 信息时，可以通过函数 $size(T)$ 来获取：

$$size(T) = \begin{cases} 0 & : & T = \phi \\ ? & : & T = leaf(x) \\ s & : & T = tree(s, F, M, R) \end{cases}$$

若树为空，则 size 为 0；若树可以表示为 $tree(s, F, M, R)$ 则 size 为 s；但是当树只有一片叶子时他的 size 是什么？它是 1 么？答案是否定的。只有当 $T = leaf(a)$ 并且 a 不是一个节点而是一个元素时 size 才等于 1。其余情况下，size 都不为 1，因为 a 可以是一个节点类型。因此我们在上面的等式中暂时放置了一个“?”。

正确的方式是通过某种形式 size 函数调用来获取信息。

$$size(T) = \begin{cases} 0 & : & T = \phi \\ size'(x) & : & T = leaf(x) \\ s & : & T = tree(s, F, M, R) \end{cases} \tag{12.37}$$

注意，这不是一个递归调用。$size \neq size'$，函数 $size'$ 的参数或者是一个 2-3 树，或者是一个普通的元素。为了统一这两种情况，我们可以将唯一的元素放入到一个节点中。这样就可以用一致的方式（节点和 size）来表示所有情况。下面的 Haskell 例子程序修改了节点的定义：

```
data Node a = Br Int [a]
```

ANSI C 例子程序的修改如下：

```
struct Node {
    Key key;
    struct Node* children;
    int size;
};
```

例子程序中，我们将 union 改为了 struct。如果节点不是叶子，key 将会带来一些额外的空间占用。

设函数 $tr(s, L)$ 从一个 size 参数 s 和一个列表 L，创建一个节点（或者是一个元素的叶子，或者是一棵 2-3 树），下面列出了一些例子：

$$\begin{array}{ll} tr(1, \{x\}) & \text{只有一个元素的树} \\ tr(2, \{x, y\}) & \text{含有两个元素的 2-3 树} \\ tr(3, \{x, y, z\}) & \text{含有 3 个元素的 2-3 树} \end{array}$$

这样，$size'$ 函数的实现就可以返回节点的 size 信息。我们有 $size'(tr(s, L)) = s$。

将元素 x 放入树中可以通过调用函数 $tr(1, \{x\})$ 来实现，我们可以定义下面的辅助函数 $wrap$ 和 $unwrap$：

$$\begin{aligned} wrap(x) &= tr(1, \{x\}) \\ unwrap(n) &= x \quad : \quad n = tr(1, \{x\}) \end{aligned} \tag{12.38}$$

现在 front 手指和 rear 手指都变成了节点的列表。为了计算手指的 size，我们可以提供一个 $size''(L)$ 函数，它把列表中每个节点的 size 加起来。记 $L = \{a_1, a_2, ...\}$、$L' = \{a_2, a_3, ...\}$。

$$size''(L) = \begin{cases} 0 & : \quad L = \phi \\ size'(a_1) + size''(L') & : \quad otherwise \end{cases} \tag{12.39}$$

也可以用一些高阶函数来定义 $size''(L)$。例如：

$$size''(L) = sum(map(size', L)) \tag{12.40}$$

我们也可以将若干节点组成的列表转换成更深一级的 2-3 树，或进行相反的转换：

$$\begin{aligned} wraps(L) &= tr(size''(L), L) \\ unwraps(n) &= L \quad : \quad n = tr(s, L) \end{aligned} \tag{12.41}$$

下面的 Haskell 例子程序实现了这些辅助函数。

```
size (Br s _) = s

sizeL = sum ∘ (map size)

sizeT Empty = 0
sizeT (Lf a) = size a
sizeT (Tr s _ _ _) = s
```

下面是 wrap 和 unwrap 辅助函数。我们省略了它们的类型定义。

```
wrap x = Br 1 [x]
unwrap (Br 1 [x]) = x
wraps xs = Br (sizeL xs) xs
unwraps (Br _ xs) = xs
```

在命令式环境中，我们可以通过结构中的变量获得节点和树的 size 信息。下面的算法将若干节点的 size 相加。

function SIZE-NODES(L)
  $s \leftarrow 0$
  for $\forall n \in L$ **do**
    $s \leftarrow s+$ SIZE(n)
  return s

下面的 Python 例子程序使用了标准库中提供的 `sum()` 和 `map()` 实现了这一操作。

```
def sizeNs(xs):
    return sum(map(lambda x: x.size, xs))
```

在命令式环境中，我们通常使用 NIL 来代表空树，可以提供一个辅助函数来统一计算非空树和空树的 size。

function SIZE-TR(T)
  if $T =$ NIL **then**
    return 0
  else
    return SIZE(T)

增加 size 信息后引入的改动

我们前面给出的算法也要针对 size 信息，做相应的修改。例如 $insertT$ 函数会先将元素放入一个节点后再插入。

$$insertT(x, T) = insertT'(wrap(x), T) \tag{12.42}$$

相应的 Haskell 例子程序修改如下：

```
cons a t = cons' (wrap a) t
```

元素 x 被放入节点后，节点的 size 为 1。此前给出插入算法中，函数 $tree(F, M, R)$ 从一个 front 手指，中间部分的子树，和一个 rear 手指构造一棵手指树。我们现在需

要从这三个参数中获得 size 信息，并累加起来存入构造好的树中。

$$tree'(F, M, R) = \begin{cases} fromL(F) & : & M = \phi \wedge R = \phi \\ fromL(R) & : & M = \phi \wedge F = \phi \\ tree'(unwraps(F'), M', R) & : & F = \phi, (F', M') = extractT'(M) \\ tree'(F, M', unwraps(R')) & : & R = \phi, (M', R') = removeT'(M) \\ tree(s, F, M, R) & : & otherwise \end{cases} \tag{12.43}$$

其中 $s = size''(F) + size(M) + size''(R)$ 是累加后的 size 信息。函数 $fromL()$ 将一个节点的列表转换为一棵手指树，它逐一将节点插入到一棵空树中。

$$fromL(L) = foldR(insertT', \phi, L)$$

当然，我们也可以不用 fold，而用递归的方法实现它。

上述算法中的最后一个情况是最简单的。若 F、M、R 都不为空，就分别取出这三部分的 size，相加到一起，并通过调用 $tree(s, F, M, R)$ 存入新构造好的树中。若中间部分的子树和任一手指都为空，算法就将另一个非空手指中的节点依次取出，插入到一棵空树中。如果中间部分的子树不为空，但是存在一个手指为空，算法就从中间部分“借”一个节点。如果 front 手指为空，就从中间部分的头部取出第一个节点作为借来的节点；若 rear 手指为空，就从中间部分的尾部取出一个节点作为借来的节点。然后算法将这个借”的节点 unwrap 成一个列表，并递归调用 $tree'()$ 函数来构造结果。

下面的 Haskell 例子程序实现了这一算法。

```
tree f Empty [] = foldr cons' Empty f
tree [] Empty r = foldr cons' Empty r
tree [] m r = let (f, m') = uncons' m in tree (unwraps f) m' r
tree f m [] = let (m', r) = unsnoc' m in tree f m' (unwraps r)
tree f m r = Tr (sizeL f + sizeT m + sizeL r) f m r
```

算法 $insertT'()$ 可以使用 $tree'()$ 定义如下：

$$insertT'(x, T) = \begin{cases} leaf(x) & : & T = \phi \\ tree'(\{x\}, \phi, \{y\}) & : & T = leaf(x) \\ tree'(\{x, x_1\}, insertT'(& : & T = tree(s, \{x_1, x_2, \\ \quad wraps(\{x_2, x_3, x_4\}), M), R) & & \quad x_3, x_4\}, M, R) \\ tree'(\{x\} \cup F, M, R) & : & otherwise \end{cases} \tag{12.44}$$

下面的 Haskell 例子程序实现了这一算法。

```
cons' a Empty = Lf a
cons' a (Lf b) = tree [a] Empty [b]
cons' a (Tr _ [b, c, d, e] m r) = tree [a, b] (cons' (wraps [c, d, e]) m) r
cons' a (Tr _ f m r) = tree (a:f) m r
```

命令式算法也需要做相应的修改，例如在向手指树的头部插入元素时，需要一边遍历，一边更新 size 信息。

function PREPEND-NODE(n, T)
  $r \leftarrow$ TREE()
  $p \leftarrow r$
  CONNECT-MID(p, T)
  while FULL?(FRONT(T)) **do**
    $F \leftarrow$ FRONT(T)
    FRONT(T) $\leftarrow \{n, F[1]\}$
    SIZE(T) $\leftarrow$ SIZE(T) + SIZE(n) ▷ update size
    $n \leftarrow$ NODE()
    CHILDREN(n) $\leftarrow F[2..]$
    $p \leftarrow T$
    $T \leftarrow$ MID(T)
  if $T =$ NIL **then**
    $T \leftarrow$ TREE()
    FRONT(T)$\leftarrow \{n\}$
  else if $\mid$ FRONT(T) $\mid = 1 \wedge$ REAR(T) $= \phi$ **then**
    REAR(T) $\leftarrow$ FRONT(T)
    FRONT(T) $\leftarrow \{n\}$
  else
    FRONT(T) $\leftarrow \{n\} \cup$ FRONT(T)
  SIZE(T) $\leftarrow$ SIZE(T) + SIZE(n) ▷ update size
  CONNECT-MID(p, T) $\leftarrow T$
  return FLAT(r)

下面的 Python 例子代码实现了这一改变。

```
def prepend_node(n, t):
    root = prev = Tree()
    prev.set_mid(t)
    while frontFull(t):
        f = t.front
        t.front = [n] + f[:1]
        t.size = t.size + n.size
        n = wraps(f[1:])
        prev = t
        t = t.mid
    if t is None:
        t = leaf(n)
    elif len(t.front)==1 and t.rear == []:
        t = Tree(n.size + t.size, [n], None, t.front)
    else:
        t = Tree(n.size + t.size, [n]+t.front, t.mid, t.rear)
```

```
    prev.set_mid(t)
    return flat(root)
```

例子代码中，树的构造函数也做了修改以便接受 size 作为第一个参数。而 `leaf` 辅助函数不仅从一个节点构造树，还将正确的 size 设置好。n

简单起见，我们不再解释 $extractT$、$appendT$、$removeT$、和 $concat$ 算法中需要针对 size 的修改，这些内容留给读者作为练习。

在指定位置分割手指树

增加 size 信息后，给定一个位置，可以很容易地通过树搜索定位到相应的节点。手指树由三部分组成 F、M、和 R，并且是递归结构。我们可以进一步根据给定的位置 i，把它分割成三个部分：左侧、位置 i 上的节点、和右侧。

我们拥有 F、M、和 R 的 size 信息。记这三部分的 size 分别为：S_f、S_m、和 S_r。如果给定的位置 $i \leq S_f$，则节点在 F 中，我们接下来在 F 中继续查找；如果 $S_f < i \leq S_f + S_m$，则节点在 M 中，我们需要递归在 M 中搜索；否则，节点一定在 R 中，我们接下来在 R 中查找。

如果忽略树为空的错误处理，则只存在一种边界情况。

$$splitAt(i, T) = \begin{cases} (\phi, x, \phi) & : \quad T = leaf(x) \\ ... & : \quad otherwise \end{cases}$$

如果对叶子进行分割，则左右部分都为空，叶子中的节点就是结果。

递归的情况根据 i 的大小又分为三种子情况。设函数 $splitAtL(i, L)$ 在位置 i 上将节点列表分割成三部分：$(A, x, B) = splitAtL(i, L)$，其中 x 为 L 中第 i 个节点，A 是 i 前的子列表，而 B 是 i 后的子列表。

$$splitAt(i, T) = \begin{cases} (\phi, x, \phi) & : \quad T = leaf(x) \\ (fromL(A), x, tree'(B, M, R)) & : \quad i \leq S_f \\ (tree'(F, M_l, A), x, tree'(B, M_r, R)) & : \quad S_f < i \leq S_f + S_m \\ (tree'(F, M, A), x, fromL(B)) & : \quad otherwise \end{cases} \tag{12.45}$$

在上式第二种情况中，有 $(A, x, B) = splitAtL(i, F)$；而在最后一种情况中，这一关系为：$(A, x, B) = splitAtL(i - S_f - S_m, R)$；比较复杂的是第三种情况，其中的 M_l、x、M_r、A、B 的计算如下：

$$\begin{aligned} (M_l, t, M_r) &= splitAt(i - S_f, M) \\ (A, x, B) &= splitAtL(i - S_f - size(M_l), unwraps(t)) \end{aligned}$$

函数 $splitAtL$ 实际上进行了线性遍历,由于列表的长度有限,且不超过 2-3 树的分

支数目限制。因此性能仍然是常数时间 $O(1)$ 的。记 $L = \{x_1, x_2, ...\}$、$L' = \{x_2, x_3, ...\}$。

$$splitAtL(i, L) = \begin{cases} (\phi, x_1, \phi) & : & i = 0 \wedge L = \{x_1\} \\ (\phi, x_1, L') & : & i < size'(x_1) \\ (\{x_1\} \cup A, x, B) & : & otherwise \end{cases} \tag{12.46}$$

其中

$$(A, x, B) = splitAtL(i - size'(x_1), L')$$

分割的解法是典型的分而治之策略。算法的性能取决于中间部分子树的递归搜索，其他情况下都是线性时间。递归的深度和树的高度 h 成比例，因此算法的性能为 $O(h)$。由于树是平衡的（使用 2-3 树，且所有的插入、删除等操作都维持树的平衡），所以 $h = O(\lg n)$，其中 n 是树中存储的元素数目。分割算法的整体性能为 $O(\lg n)$。

下面的 Haskell 例子程序给出了 $splitAtL$ 算法的实现。

```
splitNodesAt 0 [x] = ([], x, [])
splitNodesAt i (x:xs) | i < size x = ([], x, xs)
                      | otherwise = let (xs', y, ys) = splitNodesAt (i-size x) xs
                                    in (x:xs', y, ys)
```

由于 Haskell 的标准库中已经定义了同样名字的 `splitAt` 函数，为了避免冲突，我们将名称改为 `splitAt'`。

```
splitAt' _ (Lf x) = (Empty, x, Empty)
splitAt' i (Tr _ f m r)
    | i < szf = let (xs, y, ys) = splitNodesAt i f
                in ((foldr cons' Empty xs), y, tree ys m r)
    | i < szf + szm = let (m1, t, m2) = splitAt' (i-szf) m
                          (xs, y, ys) = splitNodesAt (i-szf - sizeT m1) (unwraps t)
                      in (tree f m1 xs, y, tree ys m2 r)
    | otherwise = let (xs, y, ys) = splitNodesAt (i-szf -szm) r
                  in (tree f m xs, y, foldr cons' Empty ys)
    where
      szf = sizeL f
      szm = sizeT m
```

随机访问

使用分割算法,我们可以很容易地实现性能为 $O(\lg n)$ 的随机访问。令函数 $mid(x)$ 返回一个三元组的第 2 个部分，$left(x)$ 和 $right(x)$ 分别返回第 1 和第 3 部分。

$$getAt(S, i) = unwrap(mid(splitAt(i, S))) \tag{12.47}$$

我们首先在位置 i 将序列分成 3 部分，然后取出返回的节点，并得到其中的元素。如果希望修改序列中的第 i 个元素，我们首先用 i 来分割，然后将中间部分替换成要修改的值，最后再使用连接操作将这三部分合并起来。

$$setAt(S, i, x) = concat(L, insertT(x, R)) \tag{12.48}$$

其中

$$(L, y, R) = splitAt(i, S)$$

更进一步，我们还可以实现 $removeAt(S, i)$ 算法，从一个序列 S 中删除第 i 个元素。我们首先用位置 i 分割，将节点中的元素返回，然后将左侧和右侧部分连接成一棵新手指树。

$$removeAt(S, i) = (unwrap(y), concat(L, R)) \tag{12.49}$$

下面的 Haskell 例子程序实现了这些操作。

```
getAt t i = unwrap x where (_, x, _) = splitAt' i t
setAt t i x = let (l, _, r) = splitAt' i t in concat' l (cons x r)
removeAt t i = let (l, x, r) = splitAt' i t in (unwrap x, concat' l r)
```

命令式随机访问

在命令式环境中，我们可以直接修改树中的值，因此可以不用分割而直接实现 GET-AT(T, i) 和 SET-AT(T, i, x) 算法。我们先实现一个通用算法，可以在给定位置执行指定的操作。下面的算法接受三个参数，一棵手指树 T，一个从 0 开始的位置索引 i，以及一个函数 f，用以对位置 i 上的元素实施操作。

function APPLY-AT(T, i, f)
  while SIZE$(T) > 1$ **do**
    $S_f \leftarrow$ SIZE-NODES(FRONT(T))
    $S_m \leftarrow$ SIZE-TR(MID(T))
    if $i < S_f$ **then**
      return LOOKUP-NODES(FRONT(T), i, f)
    else if $i < S_f + S_m$ **then**
      $T \leftarrow$ MID(T)
      $i \leftarrow i - S_f$
    else
      return LOOKUP-NODES(REAR(T), $i - S_f - S_m$, f)
  $n \leftarrow$ FIRST-LF(T)
  $x \leftarrow$ ELEM(n)
  ELEM$(n) \leftarrow f(x)$
  return x

算法本质上是一个分而治之的树搜索。它不断检查当前的树直到树的 size 为 1（可以通过是否是叶子来进行判断么？请考虑后面练习中的不规则树）。每次循环，我们都检查 i、front 手指的 size，和中间部分子树的 size 之间的关系。

如果索引 i 小于 front 手指的 size，则节点在 front 手指中。算法就调用一个子过程在 front 手指中查找；如果索引不比 front 手指的 size 小，但是比加上中间部分

子树的 size 的结果小，则节点在中间部分，算法就从 i 中减去 front 手指的 size，然后继续遍历中间部分的子树；否则，说明节点在 rear 手指中，算法调用子过程在其中查找。

循环结束后，我们得到一个节点（可能是一个复合节点），待查找的元素存储于这一节点的第一个叶子中。我们可以将它取出，然后对其执行函数 f，并将结果存回树中。

算法返回执行 f 前的元素作为最终结果。

接下来我们需要实现算法 LOOKUP-NODES(L, i, f)。它接受三个参数：一个节点列表，一个位置索引，和一个待执行的函数。我们可以逐一检查列表中的每个节点，如果节点为叶子，并且位置索引为 0，我们恰巧到达了指定位置。我们在这个叶子的元素上执行函数，并将此前的元素值返回；否则，我们需要比较节点的 size 和位置索引，以决定是否仅需在这个节点中搜索。

function LOOKUP-NODES(L, i, f)
  loop
    for $\forall n \in L$ **do**
      if n is leaf $\wedge i = 0$ **then**
        $x \leftarrow$ ELEM(n)
        ELEM(n) $\leftarrow f(x)$
        return x
      if $i <$ SIZE(n) **then**
        $L \leftarrow$ CHILDREN(n)
        break
      $i \leftarrow i -$ SIZE(n)

下面的 Python 例子程序实现了这一算法。

```
def applyAt(t, i, f):
    while t.size > 1:
        szf = sizeNs(t.front)
        szm = sizeT(t.mid)
        if i < szf:
            return lookupNs(t.front, i, f)
        elif i < szf + szm:
            t = t.mid
            i = i - szf
        else:
            return lookupNs(t.rear, i - szf - szm, f)
    n = first_leaf(t)
    x = elem(n)
    n.children[0] = f(x)
    return x

def lookupNs(ns, i, f):
    while True:
        for n in ns:
```

```
        if n.leaf and i == 0:
            x = elem(n)
            n.children[0] = f(x)
            return x
        if i < n.size:
            ns = n.children
            break
        i = i - n.size
```

通过将某些特殊函数传入这一通用算法，我们就可以实现 Get-At 和 Set-At 操作。

function Get-At(T, i)
  return Apply-At($T, i, \lambda_x.x$)

function Set-At(T, i, x)
  return Apply-At($T, i, \lambda_y.x$)

我们传入 id 函数来获取指定位置的元素，它并不改变元素的值；通过传入常数函数，我们可以实现设置，它忽略元素以前的值，而将传入的值作为新结果。

命令式分割

在命令式环境下，仅仅实现 Apply-At 算法还不够，我们还需要能删除指定位置的元素。

此前我们介绍的所有命令式手指树算法都只执行一轮自顶向下的操作。由于不需要自底向上进行回溯。所以，父节点到目前为止还没有派上用场。

使用父节点可以容易地实现分割操作。我们首先沿着中间部分的子树执行一轮自顶向下的遍历，直到分割位置落入 front 手指或者 rear 手指。此后，我们沿着父节点分别向上回溯两棵分割树以填入相应的内容。

function Split-At(T, i)
  $T_1 \leftarrow$ Tree()
  $T_2 \leftarrow$ Tree()
  while $S_f \leq i < S_f + S_m$ **do** ▷ 自顶向下遍历
    $T_1' \leftarrow$ Tree()
    $T_2' \leftarrow$ Tree()
    Front(T_1') $\leftarrow$ Front(T)
    Rear(T_2') $\leftarrow$ Rear(T)
    Connect-Mid(T_1, T_1')
    Connect-Mid(T_2, T_2')
    $T_1 \leftarrow T_1'$
    $T_2 \leftarrow T_2'$
    $i \leftarrow i - S_f$

```
    T ← Mid(T)
  if i < S_f then
    (X, n, Y) ← Split-Nodes(Front(T), i)
    T'_1 ← From-Nodes(X)
    T'_2 ← T
    Size(T'_2) ← Size(T) - Size-Nodes(X) - Size(n)
    Front(T'_2) ← Y
  else if S_f + S_m ≤ i then
    (X, n, Y) ← Split-Nodes(Rear(T), i − S_f − S_m)
    T'_2 ← From-Nodes(Y)
    T'_1 ← T
    Size(T'_1) ← Size(T) - Size-Nodes(Y) - Size(n)
    Rear(T'_1) ← X
  Connect-Mid(T_1, T'_1)
  Connect-Mid(T_2, T'_2)
  i ← i− Size-Tr(T'_1)
  while n is NOT leaf do                          ▷ 自底向上回溯
    (X, n, Y) ← Split-Nodes(Children(n), i)
    i ← i− Size-Nodes(X)
    Rear(T_1) ← X
    Front(T_2) ← Y
    Size(T_1) ← Sum-Sizes(T_1)
    Size(T_2) ← Sum-Sizes(T_2)
    T_1 ← Parent(T_1)
    T_2 ← Parent(T_2)
  return (Flat(T_1), Elem(n), Flat(T_2))
```

算法首先创建两棵树 T_1 和 T_2 来保存分割的结果。它们的含义都是“ground”树，为结果树的父节点。第一轮遍历是自顶向下的。令 S_f 和 S_m 分别是 front 手指和中间部分子树的 size。如果待分割的位置落在中间部分的子树中，我们就使用 T 的 front 手指作为新建的 T'_1 的 front 手指；并且复用 T 的 rear 手指作为 T'_2 的 rear 手指。此时，我们还不能设置 T'_1 和 T'_2 的其他部分，它们仍然为空，我们将在此后填入必要的信息。然后，我们将 T_1 和 T'_1 连接起来，使得后者成为前者的中间部分子树；同样我们把 T_2 和 T'_2 连接起来。最后，我们从分割位置中减去 front 手指的 size，然后继续沿着中间部分的子树遍历。

第一轮遍历结束后，我们到达一个位置，分割要么发生在 front 手指，要么发生在 rear 手指。在手指中分割会产生一个三元组，第一部分和第三部分为分割位置前后的子列表，第二部分是包含指定位置元素的节点。两个手指本质上都是 2-3 树，它

们都最多含有 3 个节点，节点分割算法可以通过线性查找来完成。

function SPLIT-NODES(L, i)
  for $j \in [1,$ LENGTH(L) $]$ **do**
    if $i <$ SIZE($L[j]$) **then**
      return $(L[1...j-1], L[j], L[j+1...$ LENGTH(L) $])$
    $i \leftarrow i-$ SIZE($L[j]$)

接下来，我们从三元组创建两个新的树 T_1' 和 T_2'，然后将它们连接起来作为 T_1 和 T_2 的最终中间子树。

此后，我们需要执行自底向上的回溯。沿着结果树填入所有尚为空的部分。

我们对三元组的第二部分，也就是节点，执行循环。直到它变为一个叶子。每次循环我们不断将节点的子树用更新的位置 i 进行分割。分割结果的第一部分子列表用于填入 T_1 作为 rear 手指；分割结果的另一部分子列表用于填入 T_2 作为 front 手指。此后，由于手指树的三个部分：front 手指、中间部分子树、和 rear 手指都完整填入了，我们就可以计算出三部分的 size 并相加，将结果作为树的 size。

function SUM-SIZES(T)
  return SIZE-NODES(FRONT(T)) + SIZE-TR(MID(T)) + SIZE-NODES(REAR(T))

接着，迭代继续沿着 T_1 和 T_2 的父节点进行。最后需要实现的算法是 FROM-NODES(L)。它从一组节点创建一棵手指树。我们可以逐一将节点插入到一棵空树中实现它。我们把它作为练习留给读者。

下面的 Python 例子程序实现了分割算法。

```
def splitAt(t, i):
    (t1, t2) = (Tree(), Tree())
    while szf(t) ≤ i and i < szf(t) + szm(t):
        fst = Tree(0, t.front, None, [])
        snd = Tree(0, [], None, t.rear)
        t1.set_mid(fst)
        t2.set_mid(snd)
        (t1, t2) = (fst, snd)
        i = i - szf(t)
        t = t.mid

    if i < szf(t):
        (xs, n, ys) = splitNs(t.front, i)
        sz = t.size - sizeNs(xs) - n.size
        (fst, snd) = (fromNodes(xs), Tree(sz, ys, t.mid, t.rear))
    elif szf(t) + szm(t) ≤ i:
        (xs, n, ys) = splitNs(t.rear, i - szf(t) - szm(t))
        sz = t.size - sizeNs(ys) - n.size
        (fst, snd) = (Tree(sz, t.front, t.mid, xs), fromNodes(ys))
    t1.set_mid(fst)
    t2.set_mid(snd)

    i = i - sizeT(fst)
    while not n.leaf:
```

```
        (xs, n, ys) = splitNs(n.children, i)
        i = i - sizeNs(xs)
        (t1.rear, t2.front) = (xs, ys)
        t1.size = sizeNs(t1.front) + sizeT(t1.mid) + sizeNs(t1.rear)
        t2.size = sizeNs(t2.front) + sizeT(t2.mid) + sizeNs(t2.rear)
        (t1, t2) = (t1.parent, t2.parent)

    return (flat(t1), elem(n), flat(t2))
```

下面的例子程序将一个节点的列表在指定位置分割开。

```
def splitNs(ns, i):
    for j in range(len(ns)):
        if i < ns[j].size:
            return (ns[:j], ns[j], ns[j+1:])
        i = i - ns[j].size
```

使用分割算法，就可以方便地实现删除操作了。我们首先执行分割，然后将结果中的两棵树连接成一棵，并将指定位置的元素返回。

function REMOVE-AT(T, i)
  $(T_1, x, T_2) \leftarrow$ SPLIT-AT(T, i)
  return (x, CONCAT(T_1, T_2))

练习 12.6

1. 另外一种实现 $insertT'$ 的方法是直接将 size 加 1，这样我们就无需使用 $tree'$ 函数。请实现这一方法。

2. 参考 $insertT'$ 的实现，完成下面的算法（分别给出函数式和命令式实现）：$extractT'$、$appendT'$、$removeT'$、和 $concat'$。而 $head$、$tail$、$init$、和 $last$ 保持不变。

3. 在命令式算法 APPLY-AT 中，我们检查当前树的 size 是否比 1 大。为什么不能检查当前的节点是否为叶子？两种方法有何区别?

4. 选择一门命令式语言，实现 FROM-NODES(L) 算法。可以使用循环或者从右侧 fold。

12.7 小结

虽然我们未能给出一个在常数时间 $O(1)$ 随机访问的纯函数式列表，但是最终的手指树数据结构实现了一个总体表现良好的序列。在头部和尾部的操作的性能为分摊时间 $O(1)$，可以在对数时间内连接两个序列，或者在任何位置将序列分割。命令

式环境中的纯数组和函数式环境中的列表都无法同时满足这些要求。某些函数式编程环境的标准库也提供了这样的序列实现 [67]。

如本章的题目所说，我们介绍了函数式环境和命令式环境中的最后一个初等数据结构。我们可以使用它们来解决一些典型问题了。

例如，当实现一个 MTF(move-to-front) 的编码算法时 [68]，就可以使用本章介绍的序列。

$$mtf(S, i) = \{x\} \cup S'$$

其中 $(x, S') = removeAt(S, i)$。

接下来的章节中，我们将介绍一些典型的分而治之的排序算法，包括快速排序，归并排序以及它们的变形；然后我们介绍一些初等搜索算法，包括一些字符串匹配算法。

第十三章　分而治之，快速排序和归并排序

13.1　简介

人们已经证明，基于比较的排序算法的最佳性能为 $O(n \lg n)$[51]。本章中，我们将要介绍两种分而治之的排序算法。它们的性能都可达到 $O(n \lg n)$。一种是快速排序，是最常用的排序算法。快速排序被广泛研究，很多编程环境的标准库都采用某种形式的快速排序作为通用排序工具。

在本章中，我们首先介绍快速排序的基本思想，它是一种典型的分而治之策略。我们会解释若干变形形式，并分析在一些特殊情况下，快速排序为什么无法均衡地分割序列，因而表现不佳。

为了解决不均衡分割的问题，我们接着会介绍归并排序，它能保证在任何情况下序列都被均分。我们还会介绍归并排序的若干变形形式，包括自然归并排序，和自底向上的归并排序。

13.2　快速排序

考虑幼儿园的老师安排小朋友们按照身高站成一队。最矮的小朋友站在最左侧，最高的小朋友站在最右侧。老师要如何给出指示，使得小朋友们能自己站好呢？

有很多方法可以做到，其中就包括快速排序的方法：

1. 第一个小朋友举起手。所有比这个小朋友矮的都站到他的左侧去；所有比他高的站到他的右侧去；
2. 所有站到左侧的小朋友重复这一步骤；所有站到右侧的小朋友也重复这一步骤。

假设一组小朋友的身高为(单位是厘米):$\{102, 100, 98, 95, 96, 99, 101, 97\}$。表13.1描述了他们按照上述方法站队的过程。

最开始的时候，身高为 102 厘米的第一个小朋友举手。我们称这个小朋友为 pivot，并用下划线标记他。恰巧这个小朋友的身高是最高的。因此所有其他人都站到他的左侧，如表中第二行所示。此时，身高为 102 厘米的小朋友站到了最终应站的位置，所

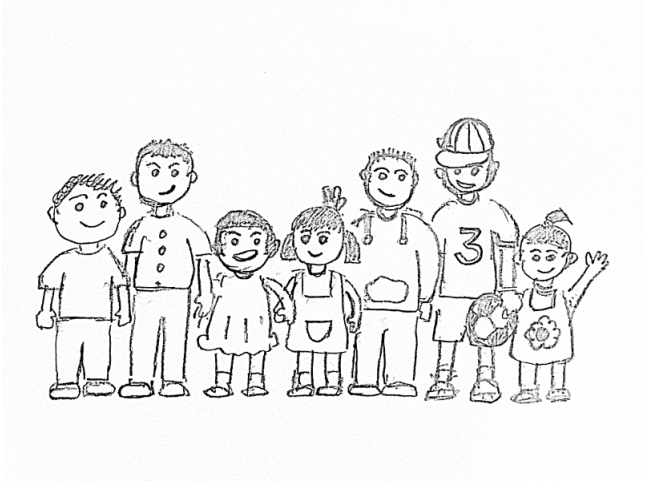


图 13.1: 安排小朋友们站成一队

以我们用引号把他括起来。接下来身高为 100 里面的小朋友举手，因此，身高为 98、95、96、和 99 厘米的小朋友站到了他的左侧，而只有一名身高为 101 厘米的小朋友身高比 pivot 高，所以他站到了右侧。表中的第三行给出了此时的状态。然后，身高为 98 厘米的小朋友成为了左侧的 pivot；而身高为 101 厘米的小朋友成为了右侧的 pivot。但是身高 101 厘米为 pivot 的那组小朋友只有他一个人，因此无需继续排序了。他站立的位置就是最终的位置。我们重复同样的方法，直到所有人都站到最终位置。

102	100	98	95	96	99	101	97
100	98	95	96	99	101	97	‘102’
98	95	96	99	97	‘100’	101	‘102’
95	96	97	‘98’	99	‘100’	‘101’	‘102’
‘95’	96	97	‘98’	‘99’	‘100’	‘101’	‘102’
‘95’	‘96’	97	‘98’	‘99’	‘100’	‘101’	‘102’
‘95’	‘96’	‘97’	‘98’	‘99’	‘100’	‘101’	‘102’

表 13.1: 一组小朋友按身高站队的过程

13.2.1 基本形式

归纳述步骤可以得到快速排序的递归描述。对序列 L 进行排序时：

- 若 L 为空，则排序结果明显为空。这是边界情况；
- 否则，在 L 中任选一个元素作为 pivot，然后递归地将 L 中不大于 pivot 的元素排序，将结果置于 pivot 的左侧，同时 递归地将所有大于 pivot 的元素排序，

将结果置于 pivot 的右侧。

这里我们强调了“同时”，而不是“然后”。也就是说，左右两侧的递归排序是可以同时并行进行的。我们后面会再次讨论有关并行的内容。

快速排序由 C. A. R. Hoare 在 1960 年提出 [51]、[78]。这里给出的描述是最基本的一种。它并没有明确解释如何选择 pivot。我们稍后会看到 pivot 的选取会直接影响到排序的性能。

最简单的方法是总选择第一个元素作为 pivot。这样就可以将快速排序形式化为下面的公式：

$$sort(L) = \begin{cases} \phi & : \quad L = \phi \\ sort(\{x | x \in L', x \leq l_1\}) \cup \{l_1\} \cup sort(\{x | x \in L', l_1 < x\}) & : \quad otherwise \end{cases} \tag{13.1}$$

其中 l_1 是非空序列 L 中的第一个元素，而 L' 包含除 l_1 外的剩余部分 $\{l_2, l_3, ...\}$。这里我们使用了 Zermelo Frankel 表达式（简称为 ZF 表达式）[1]，也称为 list comprehension。一个 ZF 表达式 $\{a | a \in S, p_1(a), p_2(a), ...\}$ 表示从集合 S 中选取使得断言 $p_1, p_2, ...$ 都为真的元素。ZF 表达式原本用于表示集合，我们将其扩展以简短地表示列表。因此允许存在重复的元素，并且不同的排列代表不同的列表。详细信息请参考本书的附录 A。

在支持 list comprehension 的编程环境中，上述公式可以直接翻译为代码。如下面的 Haskell 例子程序：

```
sort [] = []
sort (x:xs) = sort [y | y←xs, y ≤ x] ++ [x] ++ sort [y | y←xs, x < y]
```

迄今为止，这可能是最短的快速排序程序。即使引入一些中间变量，程序也仍然简洁：

```
sort [] = []
sort (x:xs) = as ++ [x] ++ bs where
    as = sort [ a | a ← xs, a ≤ x]
    bs = sort [ b | b ← xs, x < b]
```

这一基本的快速排序程序还有一些变形，例如明确使用 filter，而不是 list comprehension。如下面的 Python 例子所示：

```
def sort(xs):
    if xs == []:
        return []
    pivot = xs[0]
    as = sort(filter(lambda x : x ≤ pivot, xs[1:]))
    bs = sort(filter(lambda x : pivot < x, xs[1:]))
    return as + [pivot] + bs
```

[1] 以纪念对现代集合论贡献巨大的两位数学家。中文译作：策梅罗、弗兰克尔。

13.2.2 严格弱序

我们假设元素按照单调非递减的顺序排序。我们也可以改变算法，按照其他条件排序。这样就可以适用更多场景，在实际中，待排序的元素可能是数字、字符串、或者其他更复杂的内容（例如对一组列表排序）。

典型的方法，是把比较条件抽象成一个参数，如同此前在插入排序和选择排序的章节中所描述的。我们并不要求比较条件一定要遵从全序（total order），但是至少要满足严格弱序（strict weak order）[79]、[52]。

简单起见，我们仅仅考虑使用小于等于（不大于）作为比较条件来进行排序。

13.2.3 划分（partition）

观察前面的基本快速排序算法，会发现遍历了两次：第一次遍历获得了所有不大于 pivot 的元素，第二次遍历获得了所有大于 pivot 的元素。我们可以将他们合并成只遍历一次的划分过程。定义如下：

$$partition(p, L) = \begin{cases} (\phi, \phi) & : \quad L = \phi \\ (\{l_1\} \cup A, B) & : \quad p(l_1), (A, B) = partition(p, L') \\ (A, \{l_1\} \cup B) & : \quad \neg p(l_1) \end{cases} \tag{13.2}$$

这里的 $\{x\} \cup L$ 仅仅是一个“cons”操作（将元素链接到表头），它只需要常数时间。使用 partition，快速排序可以定义为：

$$sort(L) = \begin{cases} \phi & : \quad L = \phi \\ sort(A) \cup \{l_1\} \cup sort(B) & : \quad L \neq \phi, (A, B) = partition(\lambda_x x \leq l_1, L') \end{cases} \tag{13.3}$$

下面的 Haskell 例子程序实现了这一算法。

```
sort [] = []
sort (x:xs) = sort as ++ [x] ++ sort bs where
    (as, bs) = partition (≤ x) xs

partition _ [] = ([], [])
partition p (x:xs) = let (as, bs) = partition p xs in
    if p x then (x:as, bs) else (as, x:bs)
```

划分（partition）的概念对于快速排序至关重要。划分在其很多其他排序排序算法中也很关键。本章的最后部分会解释它如何普遍地影响着排序的思想方法。在进一步改进快速排序的划分算法前，我们先来考虑如何用命令式的方法实现原地快速排序。

在诸多的划分方法中，Lomuto[2]、[4] 给出的方法是最简单易懂的。我们稍后还会介绍其他划分方法，并展示不同的方法是如何影响性能的。

图13.2描述了这种一次遍历进行划分的方法。我们从左向右逐一处理数组中的元素。任何时候，数组都由图13.2 (a) 所示的几部分组成：

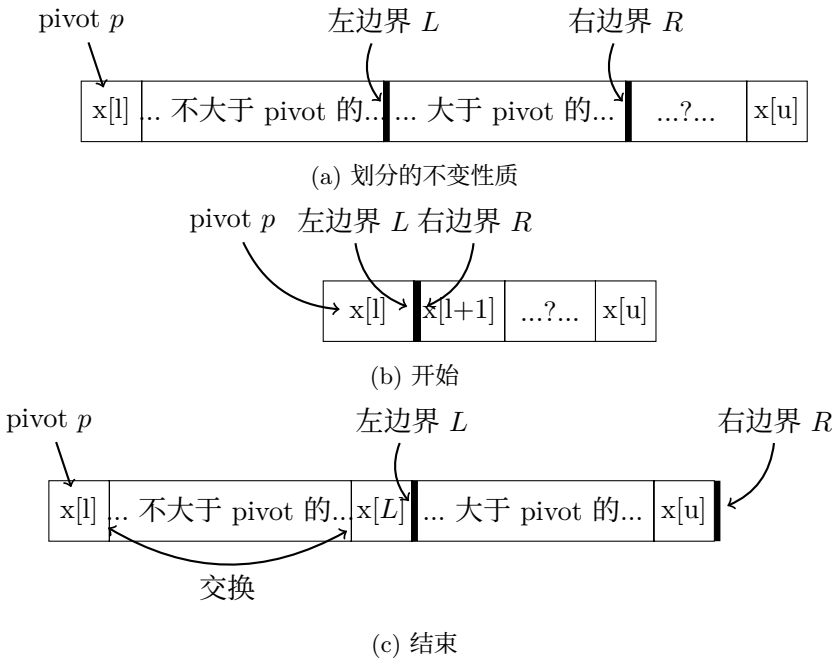


图 13.2: 使用最左边的元素作 pivot 划分一段数组

- 最左侧为 pivot，当划分过程结束时，pivot 会被移动到最终的位置；
- 一段只包含不大于 pivot 的元素的部分。这一段的右侧边界被标记为 L；
- 一段只包含大于 pivot 的元素的部分。这一段的右侧边界被标记为 R。也就是说，L 标记和 R 标记之间的元素都大于 pivot；
- R 标记后面的元素尚未被处理。这部分的元素可能大于，也可能不大于 pivot。

在划分过程开始的时候，L 标记指向 pivot，R 标记指向 pivot 后的下一个元素，如图13.2 (b) 所示。然后算法不断地向右侧移动 R 标记进行处理直到 R 标记越过数组的右侧边界。

每次迭代，都比较 R 标记指向的元素和 pivot 的大小。若大于 pivot，这一元素应该位于 L 和 R 标记之间，算法继续向前移动 R 标记以检查下一个元素；否则，说明 R 标记指向的元素小于或者等于 pivot（不大于），它应该位于 L 标记的左侧。为此，我们将 L 标记向前移动一步，然后交换 L 和 R 标记指向的元素。

当 R 标记越过最后一个元素时，所有的元素都已处理完毕。大于 pivot 的元素都被移动到了 L 标记的右侧，而其他元素位于 L 标记的左侧。此时我们需要移动 pivot 元素，使得它位于这两段的中间。为此，我们可以交换 pivot 和 L 标记指向的元素。如图13.2 (c) 中的双向箭头所示。

L 标记最终指向 pivot，它将整个的数组分成了两部分。我们将 L 标记作为划分过程的结果返回。实际中，为了方便后继处理，我们通常将 L 标记增加 1，使得它指向第一个大于 pivot 的元素。整个划分过程中，我们就地修改了数组中的内容。

划分算法可以描述如下。它接受三个参数：一个数组 A，待划分区间的上下界[2]

```
1: function PARTITION(A, l, u)
2:     p ← A[l]                                  ▷ p 为 pivot
3:     L ← l                                     ▷ 左侧标记
4:     for R ∈ [l + 1, u] do                     ▷ 对右侧标记进行迭代
5:         if ¬(p < A[R]) then                   ▷ 对于严格弱序，定义 < 比较就足够了
6:             L ← L + 1
7:             EXCHANGE A[L] ↔ A[R]
8:     EXCHANGE A[L] ↔ p
9:     return L + 1                              ▷ 返回划分的位置
```

表13.2给出了划分数组 $\{3, 2, 5, 4, 0, 1, 6, 7\}$ 的步骤。

$\underline{3}$(l)	2(r)	5	4	0	1	6	7	开始，$pivot = 3$、$l = 1$、$r = 2$
$\underline{3}$	2(l)(r)	5	4	0	1	6	7	$2 < 3$，移动 l（$r = l$）
$\underline{3}$	2(l)	5(r)	4	0	1	6	7	$5 > 3$, 继续
$\underline{3}$	2(l)	5	4(r)	0	1	6	7	$4 > 3$, 继续
$\underline{3}$	2(l)	5	4	0(r)	1	6	7	$0 < 3$
$\underline{3}$	2	0(l)	4	5(r)	1	6	7	移动 l，然后和 r 交换
$\underline{3}$	2	0(l)	4	5	1(r)	6	7	$1 < 3$
$\underline{3}$	2	0	1(l)	5	4(r)	6	7	移动 l，然后和 r 交换
$\underline{3}$	2	0	1(l)	5	4	6(r)	7	$6 > 3$，继续
$\underline{3}$	2	0	1(l)	5	4	6	7(r)	$7 > 3$，继续
1	2	0	3	5(l+1)	4	6	7	r 越过了边界，交换 pivot 和 l

表 13.2: 扫描并划分数组的步骤

下面的 ANSI C 例子程序实现了这一划分算法。

```
int partition(Key* xs, int l, int u) {
    int pivot, r;
    for (pivot = l, r = l + 1; r < u; ++r)
        if (!(xs[pivot] < xs[r])) {
            ++l;
            swap(xs[l], xs[r]);
        }
    swap(xs[pivot], xs[l]);
    return l + 1;
}
```

其中 `swap(a, b)` 可以定义为函数或者宏。ISO C++ 中 `swap(a, b)` 在标准库中以函数模板的形式提供。被交换的元素类型通过模板进行推导。我们此后不再详细解释这些语言细节。

[2]这里描述的算法和 [4] 中的略有不同，后者用待划分区间的最后一个元素作为 pivot。

使用这一划分算法，命令式的原地快速排序可以实现如下：

```
1: procedure QUICK-SORT(A, l, u)
2:     if l < u then
3:         m ← PARTITION(A, l, u)
4:         QUICK-SORT(A, l, m − 1)
5:         QUICK-SORT(A, m, u)
```

对数组进行排序时，我们传入数组的上下界，如：QUICK-SORT$(A, 1, |A|)$。其中 $l \geq u$ 用以判断数组片段为空或者只含有一个元素，这两种情况下我们都认为数组是已序的，算法直接返回而无需做任何处理。

下面的 ANSI C 例子程序给出了原地快速排序的实现。

```
void quicksort(Key* xs, int l, int u) {
    int m;
    if (l < u) {
        m = partition(xs, l, u);
        quicksort(xs, l, m - 1);
        quicksort(xs, m, u);
    }
}
```

13.2.4 函数式划分算法的小改进

在深入分析快速排序的划分算法前，我们首先可以用 fold 来实现一个小改进：只需要遍历一遍就可以完成划分的算法。读者可以参考本书附录 A 来了解 fold 的详细内容。

$$partition(p, L) = fold(f(p), (\phi, \phi), L) \tag{13.4}$$

其中函数 f 使用断言 p 来对元素和 pivot 进行比较。断言作为一个参数传入函数 f，我们称之为 f 的“柯里化”形式（Currying form），参见附录 A。另外，f 可以是 $partition$ 函数作用域内的一个词法闭包（lexical closure），它可以访问这一作用域内的断言。函数 f 不断更新划分结果内的一对列表。

$$f(p, x, (A, B)) = \begin{cases} (\{x\} \cup A, B) & : \quad p(x) \\ (A, \{x\} \cup B) & : \quad \neg p(x) \end{cases} \tag{13.5}$$

我们这里使用了模式匹配（pattern-matching）形式的定义。在不支持模式匹配的环境中，需要使用一个变量，如 P 来代表列表对 (A, B)，并使用函数来获取 P 中的两个值。

下面的 Haskell 例子程序实现了这一改进的快速排序，每次划分只需要遍历一次。

```
sort [] = []
sort (x:xs) = sort small ++ [x] ++ sort big where
  (small, big) = foldr f ([], []) xs
  f a (as, bs) = if a ≤ x then (a:as, bs) else (as, a:bs)
```

累积划分（Accumulated partition）

使用 fold 进行划分的过程，实际上是向结果列表对 (A, B) 累积的过程。若元素不大于 pivot，则它被累积到 A，否则累积到 B。我们可以将这一累积过程明确定义出来，相对于最初的基本快速排序算法，这样既可以节省空间，又利于进行尾递归优化（参见附录 A）。

$$partition(p, L, A, B) = \begin{cases} (A, B) & : \quad L = \phi \\ partition(p, L', \{l_1\} \cup A, B) & : \quad p(l_1) \\ partition(p, L', A, \{l_1\} \cup B) & : \quad otherwise \end{cases} \tag{13.6}$$

其中，若列表 L 不空，则 l_1 代表其中的第一个元素，L' 代表除第一元素外的剩余部分，形如：$L' = \{l_2, l_3, ...\}$。通过向划分函数传入比较参数，如：$\lambda_x x \leq pivot$ 即可以实现升序的排序算法。

$$sort(L) = \begin{cases} \phi & : \quad L = \phi \\ sort(A) \cup \{l_1\} \cup sort(B) & : \quad otherwise \end{cases} \tag{13.7}$$

其中 A、B 是通过上述划分函数计算出的结果。

$$(A, B) = partition(\lambda_x x \leq l_1, L', \phi, \phi)$$

累积式快速排序

观察前面快速排序定义中的递归部分可以发现，列表的连接操作 $sort(A) \cup \{l_1\} \cup sort(B)$ 需要的时间和列表的长度成比例。可以使用附录 A 中介绍的一些方法提高性能，另外，也可以将排序算法转换为累积形式。

$$sort'(L, S) = \begin{cases} S & : \quad L = \phi \\ ... & : \quad otherwise \end{cases}$$

其中 S 为累积结果。我们传入一个空的起始值来启动排序：$sort(L) = sort'(L, \phi)$。当划分完成时，需要递归地对两个子列表进行排序。我们可以先递归地将大于 pivot 的元素排序，然后将 pivot 链接到这一结果的前面。然后将链接结果作为新的“累积结果”传入后续的排序过程中。

根据这一思路，上述算法中的省略号部分可以实现如下：

$$sort'(L, S) = \begin{cases} S & : \quad L = \phi \\ sort(A, \{l_1\} \cup sort(B, ?)) & : \quad otherwise \end{cases}$$

当开始对 B 排序时，累积结果应该是什么呢？这里有一个很重要的不变性质：任何时候，累积结果 S 中总保存了迄今为止已经排序好的元素。因此，我们通过向 S 累积来对 B 排序。

$$sort'(L, S) = \begin{cases} S & : \quad L = \phi \\ sort(A, \{l_1\} \cup sort(B, S)) & : \quad otherwise \end{cases} \tag{13.8}$$

下面的 Haskell 例子程序实现了累积式快速排序算法。

```
asort xs = asort' xs []

asort' [] acc = acc
asort' (x:xs) acc = asort' as (x:asort' bs acc) where
  (as, bs) = part xs [] []
  part [] as bs = (as, bs)
  part (y:ys) as bs | y ≤ x = part ys (y:as) bs
                    | otherwise = part ys as (y:bs)
```

练习 13.1

- 选择一门命令式语言，实现递归的基本快速排序算法。
- 和命令式快速排序算法类似，除了列表为空的情况外，如果列表只含有一个元素，也可以作为边界情况处理。修改函数式算法，处理这一边界情况。
- 在累积式快速排序算法的实现中，使用了中间变量 A、B。我们可以通过重新定义划分函数，通过递归调用 sort 函数来消除中间变量。选择一门函数式编程语言，实现这一改动。

13.3 快速排序的性能分析

快速排序在实际应用中性能良好，但是给出严格的分析却并不容易。我们需要使用统计学工具来证明平均情况下的性能。

尽管如此，我们可以很直观地计算出最好情况和最坏情况下的性能。显然，最好情况发生在每次划分都能将序列均分成两段长度相同子序列时。如图13.3所示，共需要 $O(\lg n)$ 次递归调用。

总共有 $O(\lg n)$ 层递归。在第一层，进行一次划分，处理 n 个元素；在第二层，进行两次划分，每次划分处理 $n/2$ 个元素，第二层的总体执行时间为 $2O(n/2) = O(n)$。在第三层，执行划分四次，每次处理 $n/4$ 个元素，第三层的总体执行时间也是 $O(n)$ ……在最后一层，总共有 n 个片段，每个片段只含有一个元素，总处理时间也是 $O(n)$。将上述所有层的执行时间相加，得到快速排序在最好情况下的性能为 $O(n \lg n)$。

但是在最坏情况下，划分过程大部分时间都把序列分成两个很不平衡的部分。其中一部分的长度为 $O(1)$，另一部分的长度为 $O(n)$。因此递归的深度退化为 $O(n)$。如果我们用同样的图来描述，最好情况下，快速排序过程形成一棵平衡二叉树；而最坏情况下，会形成一棵很不平衡的树，每个节点都只有一棵子树，而另外一棵子树为空。二叉树退化成了一个长度为 $O(n)$ 的链表。而在每一层中，所有的元素都被处理，因此最坏情况下的性能为 $O(n^2)$，这和插入排序、选择排序的性能相当。

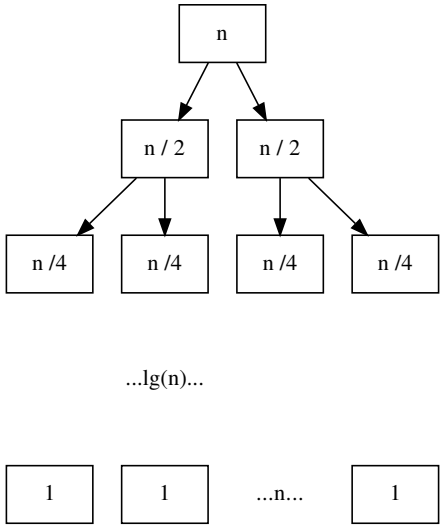


图 13.3: 最好情况下，快速排序每次将序列划分成长度相等的两部分

最坏情况在何时发生的？其中一个特殊情况是所有的元素（或大部分元素）都相同。Lomuto 的划分方法此时的表现很差。我们在下一节会介绍另外一种划分方法，可以有效解决这一问题。

另外有两种明显的序列类型可以导致最坏情况：即序列已序（升序或降序）。划分升序序列时 pivot 前的部分总是空的，而 pivot 后的部分包含所有剩余元素。划分降序序列的结果与此相反。

还有其他一些情况可以导致快速排序的性能很差。不存在一种方法可以完全避免最坏情况。我们下一节会给出一些工程方法可以把最坏情况发生的可能性降低。

13.3.1 平均情况的分析 ⋆

快速排序在平均情况下性能良好。甚至在每次划分时，总得到长度比为 1:9 的两部分，总体性能仍然为 $O(n \lg n)$[4]。

本节需要一些额外的数学知识，读者可以选择跳过。

有两种方法可以证明快速排序在平均情况下的性能。其中一种方法利用了快速排序中的比较操作的次数来考量性能 [4]。例如，在选择排序中，任何两个元素都进行了比较。而快速排序却避免了很多不必要的比较。考虑划分列表 $\{a_1, a_2, a_3, ..., a_n\}$，选择 a_1 作为 pivot，划分结果产生两个子列表 $A = \{x_1, x_2, ..., x_k\}$ 和 $B = \{y_1, y_2, ..., y_{n-k-1}\}$。在接下来的快速排序过程中，$A$ 中的任何元素，都不再和 B 中的任何元素进行比较。

记最终排序的结果为 $\{a_1, a_2, ..., a_n\}$，我们有这样的结果：若 $a_i < a_j$，当且仅当存在某一元素 a_k 满足 $a_i < a_k < a_j$，并且 a_k 在 a_i 或 a_j 之前被选为 pivot 时，我们将不再对 a_i 和 a_j 进行比较。

也就是说，a_i 与 a_j 进行比较的唯一可能是要么 a_i，要么 a_j 在所有 $a_{i+1} < a_{i+2} < ... < a_{j-1}$ 之前被选为 pivot。

令 $P(i,j)$ 代表 a_i 和 a_j 进行比较的概率，我们有：

$$P(i,j) = \frac{2}{j-i+1} \tag{13.9}$$

全部比较操作的总数可以这样得到：

$$C(n) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} P(i,j) \tag{13.10}$$

如果我们比较了 a_i 和 a_j，在接下来的快速排序中，就不再比较 a_j 和 a_i，并且元素 a_i 永远不会和自己进行比较。因此在上式中，i 的上限为 $n-1$，j 的下限为 $i+1$。

将概率代入，得：

$$\begin{aligned} C(n) &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} \\ &= \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1} \end{aligned} \tag{13.11}$$

使用调和级数 [80]。

$$H_n = 1 + \frac{1}{2} + \frac{1}{3} + ... = \ln n + \gamma + \epsilon_n$$

因此：

$$C(n) = \sum_{i=1}^{n-1} O(\lg n) = O(n \lg n) \tag{13.12}$$

我们还可以用另外一种方法证明快速排序在平均情况下的性能。考虑递归，当待排序的列表长度为 n 时，划分过程将列表分成两个部分，一部分长度为 i，另一部分长度为 $n-i-1$。划分过程需要比较 pivot 和每个元素，它自身用时 cn。因此我们有如下递归关系：

$$T(n) = T(i) + T(n-i-1) + cn \tag{13.13}$$

其中 $T(n)$ 是对长度为 n 的列表进行快速排序所用的时间。由于 i 以相同的概率在 $0, 1, ..., n-1$ 中取值，通过使用数学期望，可以得到如下结果：

$$\begin{aligned} T(n) &= E(T(i)) + E(T(n-i-1)) + cn \\ &= \frac{1}{n} \sum_{i=0}^{n-1} T(i) + \frac{1}{n} \sum_{i=0}^{n-1} T(n-i-1) + cn \\ &= \frac{1}{n} \sum_{i=0}^{n-1} T(i) + \frac{1}{n} \sum_{j=0}^{n-1} T(j) + cn \\ &= \frac{2}{n} \sum_{i=0}^{b-1} T(i) + cn \end{aligned} \tag{13.14}$$

两边同时乘以 n：

$$nT(n) = 2\sum_{i=0}^{n-1} T(i) + cn^2 \tag{13.15}$$

将 n 用 $n-1$ 替换，可以得到另外一个等式：

$$(n-1)T(n-1) = 2\sum_{i=0}^{n-2} T(i) + c(n-1)^2 \tag{13.16}$$

用式 (13.15) 减去式 (13.16) 可以消去所有的 $T(i)$，其中 $0 \le i < n-1$。

$$nT(n) = (n+1)T(n-1) + 2cn - c \tag{13.17}$$

在计算性能时，我们可以忽略掉常数时间 c。因此上式进一步变化为：

$$\frac{T(n)}{n+1} = \frac{T(n-1)}{n} + \frac{2c}{n+1} \tag{13.18}$$

我们依次用 $n-1$、$n-2$……代入 n，可以得到 $n-1$ 个等式。

$$\frac{T(n-1)}{n} = \frac{T(n-2)}{n-1} + \frac{2c}{n}$$

$$\frac{T(n-2)}{n-1} = \frac{T(n-3)}{n-2} + \frac{2c}{n-1}$$

$$\cdots$$

$$\frac{T(2)}{3} = \frac{T(1)}{2} + \frac{2c}{3}$$

将所有等式相加，消去左右两侧相同的变量，可以化简得到一个关于 n 的函数。

$$\frac{T(n)}{n+1} = \frac{T(1)}{2} + 2c\sum_{k=3}^{n+1} \frac{1}{k} \tag{13.19}$$

使用上面提到的调和级数，最终的结果为：

$$O(\frac{T(n)}{n+1}) = O(\frac{T(1)}{2} + 2c\ln n + \gamma + \epsilon_n) = O(\lg n) \tag{13.20}$$

因此

$$O(T(n)) = O(n \lg n) \tag{13.21}$$

练习 13.2

- 当有很多重复元素时，为什么 Lomuto 的方法性能会变差？

13.4 工程实践中的改进

大多数情况下快速排序性能优异。但是在最差的情况下，性能会下降到平方级别。如果待排序的数据是完全随机分布的，出现最差情况的概率会很低。尽管如此，某些常见的特殊序列却仍会引发最差情况。

本节我们介绍一些工程上常用的方法，它们或者针对某些特殊的输入数据改进划分算法来避免性能下降，或者通过改变概率分布来减小出现最差情况的可能。

13.4.1 处理重复元素的工程方法

如上一节的练习中所示，Lomuto 的划分算法不擅长处理含有很多重复元素的序列。考虑含有 n 个相等元素的特殊序列 $\{x, x, ..., x\}$，我们有两种方案来进行排序。

1. 普通的基本快速排序法：我们任意选择一个元素作为 pivot，其值为 x，这样分割后得到两个子序列，一个是 $\{x, x, ..., x\}$，包含 $n-1$ 个元素，另外一个子序列为空。接下来递归地对第一个子序列排序；这明显是一个 $O(n^2)$ 的解决方法。

2. 另外一个方法是只挑选严格小于 x 的元素，和严格大于 x 的元素进行划分。这样得到的结果是两个空序列，和 n 个等于 pivot 的元素。接下来我们递归地对只含有小于 pivot 的元素子序列和只含有大于 pivot 的元素的子序列进行排序，由于它们都为空，因此递归调用立即结束。剩下要做的就是将比 pivot 小的元素的排序结果，全部等于 pivot 的元素，和比 pivot 大的元素的排序结果连接起来。

如果所有元素都相等，第二种方法只需要 $O(n)$ 时间。这给出了划分算法的一个重要改进：相对于二分划分 (binary partition，划分成两个子序列和一个 pivot)，三分划分（ternary partition，划分成三个子序列）能更好地处理重复元素。

我们可以这样来定义三分划分快速排序（ternary quick sort）：

$$sort(L) = \begin{cases} \phi & : \quad L = \phi \\ sort(S) \cup sort(E) \cup sort(G) & : \quad otherwise \end{cases} \tag{13.22}$$

其中 S, E, G 分别是所有小于、等于、和大于 pivot 的元素组成的列表。

$$\begin{aligned} S &= \{x | x \in L, x < l_1\} \\ E &= \{x | x \in L, x = l_1\} \\ G &= \{x | x \in L, l_1 < x\} \end{aligned}$$

下面的 Haskell 例子程序实现了基本的三分快速排序算法。

```
sort [] = []
sort (x:xs) = sort [a | a←xs, a<x] ++
              x:[b | b←xs, b==x] ++ sort [c | c←xs, c>x]
```

注意，元素间的比较必须支持“小于”和“等于”操作，而普通快速排序仅仅要求“小于”比较。在性能上，基本的三分快速排序需要线性时间 $O(n)$ 将三个子列表连接起来。可以使用一个累积变量（accumulator）来改善这一性能。

令函数 $sort'(L, A)$ 表示带有累积变量的三分快速排序定义，其中 L 为待排序序列，累积变量 A 包含已排好序的部分。它最开始时为空：$sort(L) = sort'(L, \phi)$。我们可以先定义好边界条件：

$$sort'(L, A) = \begin{cases} A & : & L = \phi \\ ... & : & otherwise \end{cases}$$

对于递归情况，三分划分将序列分为三个子序列 S, E, G，其中只有 S 和 G 需要递归排序，而 E 包含全部等于 pivot 的元素，无需进一步排序了。我们可以先使用累积变量 A 对 G 进行排序，然后将排序结果连接到 E 的后面，作为新的累积变量对 S 进行排序。

$$sort'(L, A) = \begin{cases} A & : & L = \phi \\ sort(S, E \cup sort(G, A)) & : & otherwise \end{cases} \tag{13.23}$$

划分算法也可以使用累积变量来实现。这和基本的快速排序类似。注意这里我们不能只传入一个和 pivot 进行比较的断言，而需要传入两个：一个用于“小于”比较，另外一个用于“等于”判断。简单起见，这里我们传入 pivot 元素。

$$partition(p, L, S, E, G) = \begin{cases} (S, E, G) & : & L = \phi \\ partition(p, L', \{l_1\} \cup S, E, G) & : & l_1 < p \\ partition(p, L', S, \{l_1\} \cup E, G) & : & l_1 = p \\ partition(p, L', S, E, \{l_1\} \cup G) & : & p < l_1 \end{cases} \tag{13.24}$$

其中，若 L 不为空，l_1 为 L 中的第一个元素，L' 包含除 l_1 外的剩余部分。下面的 Haskell 例子程序实现了这一算法。它在划分算法的边界情况中启动递归排序。

```
sort xs = sort' xs []

sort' []     r = r
sort' (x:xs) r = part xs [] [x] [] r where
    part [] as bs cs r = sort' as (bs ++ sort' cs r)
    part (x':xs') as bs cs r | x' <  x = part xs' (x':as) bs cs r
                             | x' == x = part xs' as (x':bs) cs r
                             | x' >  x = part xs' as bs (x':cs) r
```

Richard Bird 给出了另外一个改进 [1]，它不对递归排序的结果立即执行连接操作，而是把排好的子列表放入一个列表中保存。最终再将这些子列表连接在一起。

```
sort xs = concat $ pass xs []

pass [] xss = xss
pass (x:xs) xss = step xs [] [x] [] xss where
    step [] as bs cs xss = pass as (bs:pass cs xss)
```

```
    step (x':xs') as bs cs xss | x' <  x = step xs' (x':as) bs cs xss
                               | x' == x = step xs' as (x':bs) cs xss
                               | x' >  x = step xs' as bs (x':cs) xss
```

双向划分（2-way partition）

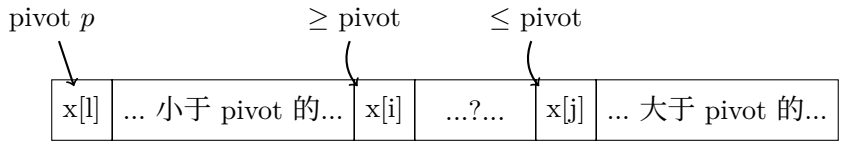
也可以用命令式的方法解决大量重复元素的问题。Robert Sedgewick 给出了一个划分方法 [69]、[2]，使用两个指针，一个从左向右移动，另一个从右向左移动。开始的时候两个指针指向数组的左右边界。

划分开始时，选择最左侧的元素作为 pivot。然后左侧指针 i 不断向右前进直到遇到一个不小于 pivot 的元素；另外[3]，右侧指针 j 不断向左扫描直到遇到一个不大于 pivot 的元素。

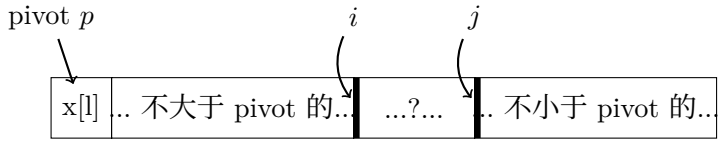
此时，所有在左侧指针 i 之前的元素都严格小于 pivot，而所有在右侧指针 j 之后的元素都严格大于 pivot。i 指向一个大于或等于 pivot 的元素；而 j 指向一个小于或等于 pivot 的元素。图13.4 (a) 描述了此时的情形。

为了将全部小于或等于 pivot 的元素划分到左侧，而其余元素划分到右侧，我们可以交换 i 和 j 指向的两个元素。然后我们恢复扫描，重复上面的步骤直到 i 和 j 相遇或者交错。

在划分的任何时刻，总保持着不变条件（invariant），即所有 i 之前的元素（包括 i 指向的元素）都不大于 pivot；而所有 j 之后的元素（包括 j 指向的元素）都不小于 pivot。i 和 j 之间的元素尚未处理。图13.4 (b) 描述了这一不变条件。



(a) 指针 i 和 j 停止前进时



(b) 划分不变条件

图 13.4: 选择最左侧的元素作为 pivot 进行划分

当左侧指针 i 和右侧指针 j 相遇或交错时，我们需要进行一次额外的交换操作，将最左侧的 pivot 元素交换到 j 指向的位置上。然后，我们对划分区间下界和 j 之间的数组片段，以及 i 和划分区间上界之间的片段进行递归排序。

这一算法可以描述如下。

[3]注意，我们没有使用“然后”一词，因为这两轮扫描可以同时并发进行。

```
1: procedure SORT(A, l, u)                                  ▷ sort range [l, u)
2:     if u − l > 1 then                                    ▷ 非平凡情况下包含 1 个以上的元素
3:         i ← l, j ← u
4:         pivot ← A[l]
5:         loop
6:             repeat
7:                 i ← i + 1
8:             until A[i] ≥ pivot                           ▷ 忽略 i ≥ u 的错误处理
9:             repeat
10:                j ← j − 1
11:            until A[j] ≤ pivot                           ▷ 忽略 j < l 的错误处理
12:            if j < i then
13:                break
14:            EXCHANGE A[i] ↔ A[j]
15:        EXCHANGE A[l] ↔ A[j]                             ▷ 移动 pivot
16:        SORT(A, l, j)
17:        SORT(A, i, u)
```

考虑所有元素都相等的极端情况，这一原地快速排序将数组划分为两段长度相等的子数组，这里发生了 $\frac{n}{2}$ 次不必要的交换操作。由于划分是平衡的，所以总体性能仍然为 $O(n \lg n)$，而没有下降到平方级别。下面的 C 语言例子程序实现了这一算法。

```
void qsort(Key* xs, int l, int u) {
    int i, j, pivot;
    if (l < u - 1) {
        pivot = i = l; j = u;
        while (1) {
            while (i < u && xs[++i] < xs[pivot]);
            while (j >=l && xs[pivot] < xs[--j]);
            if (j < i) break;
            swap(xs[i], xs[j]);
        }
        swap(xs[pivot], xs[j]);
        qsort(xs, l, j);
        qsort(xs, i, u);
    }
}
```

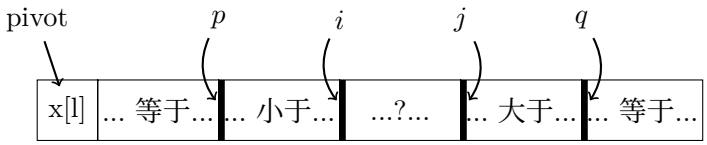
和此前介绍的 Lomuto 的划分算法相比，可以发现这一算法的元素交换操作次数更少。这是因为它跳过了那些最终位置在 pivot 正确一侧的元素不进行交换。

三路划分

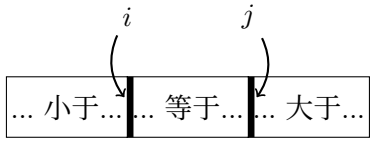
显然，我们应该避免对重复元素进行不必要的交换操作。进一步，可以利用“三分排序”（ternary sort，也称作三路划分）的思路来改进算法，所有严格小于 pivot

的元素被放入左侧的子序列片段，严格大于 pivot 的元素被放入右侧，而中间部分包含所有等于 pivot 的元素。使用三路划分，我们只需要对不等于 pivot 的元素进行递归排序。在上述的特殊情况中，由于所有的元素都相等，我们无需进行进一步的递归排序。因此整体的性能为线性时间 $O(n)$。

我们接下来需要考虑如何实现三路划分。Jon Bentley 和 Douglas McIlroy 给出了一个方法：如图13.5 (a) 所示，所有和 pivot 相等的元素最初保存在最左侧和最右侧 [70]、[71]。



(a) 三路划分的不变条件。



(b) 将和 pivot 相等的元素交换到中间部分。

图 13.5: 三路划分

扫描过程大部分和 Robert Sedgewick 给出的相似，两个指针 i 和 j 相向前进直到 i 遇到任何大于等于 pivot 的元素，并且 j 遇到任何小于等于 pivot 的元素。此时，如果 i 和 j 没有相遇或者交错，我们不仅交换它们指向的元素，同时检查被指向的元素是否和 pivot 相等，如果相等，就交换 i 和 p 指向的元素，以及 j 和 q 指向的元素。

在划分过程结束前，需要把所有等于 pivot 的元素从左右两侧交换到中间。交换的次数取决于重复元素的个数。如果所有的元素都不等，则交换次数为零，不产生任何额外的性能消耗。划分的最终结果如图13.5 (b) 所示。此后，我们只需要对“严格小于”和“严格大于”部分的子片段进行递归排序。

可以通过修改两路划分的算法进行实现。

```
1:  procedure SORT(A, l, u)
2:      if u − l > 1 then
3:          i ← l, j ← u
4:          p ← l, q ← u                          ▷ 指向相等元素的边界
5:          pivot ← A[l]
6:          loop
7:              repeat
8:                  i ← i + 1
9:              until A[i] ≥ pivot                ▷ 忽略 i ≥ u 的错误处理
10:             repeat
```

```
        j ← j − 1
    until A[j] ≤ pivot                          ▷ 忽略 j < l 的错误处理
    if j ≤ i then
        break                                   ▷ 注意和此前算法的不同
    Exchange A[i] ↔ A[j]
    if A[i] = pivot then                        ▷ 处理相等的元素
        p ← p + 1
        Exchange A[p] ↔ A[i]
    if A[j] = pivot then
        q ← q − 1
        Exchange A[q] ↔ A[j]
if i = j ∧ A[i] = pivot then                    ▷ 特殊情况
    j ← j − 1, i ← i + 1
for k from l to p do                            ▷ 将相等的元素交换到中间
    Exchange A[k] ↔ A[j]
    j ← j − 1
for k from u − 1 down-to q do
    Exchange A[k] ↔ A[i]
    i ← i + 1
Sort(A, l, j + 1)
Sort(A, i, u)
```

下面的 C 语言例子程序实现了三路划分快速排序算法。

```
void qsort2(Key* xs, int l, int u) {
    int i, j, k, p, q, pivot;
    if (l < u - 1) {
        i = p = l; j = q = u; pivot = xs[l];
        while (1) {
            while (i < u && xs[++i] < pivot);
            while (j ≥ l && pivot < xs[--j]);
            if (j ≤ i) break;
            swap(xs[i], xs[j]);
            if (xs[i] == pivot) { ++p; swap(xs[p], xs[i]); }
            if (xs[j] == pivot) { --q; swap(xs[q], xs[j]); }
        }
        if (i == j && xs[i] == pivot) { --j, ++i; }
        for (k = l; k ≤ p; ++k, --j) swap(xs[k], xs[j]);
        for (k = u-1; k ≥ q; --k, ++i) swap(xs[k], xs[i]);
        qsort2(xs, l, j + 1);
        qsort2(xs, i, u);
    }
}
```

引入三路划分后，算法逐渐变得复杂了。各种边界条件都需要进行仔细的处理。

回顾此前的 Lomuto 的划分方法，它的优势就是简单直观，我们可以考虑对它加以改进，得到一个简单的三路划分实现。

我们需要调整一下不变条件（invariant）。我们仍然选择第一个元素作为 pivot，如图13.6所示，任何时刻，左侧的片段包含严格小于 pivot 的元素；接下来的片段包含等于 pivot 的元素；最右侧的片段包含严格大于 pivot 的元素。这三个片段的边界分别为 i、k 和 j。剩余在 k 和 j 之间的部分是尚未扫描的元素。

我们从左向右逐一扫描元素，一开始时，严格小于 pivot 的部分为空；等于 pivot 的部分只包含一个元素，就是 pivot 本身。i 此时指向数组的下界，k 指向 i 的下一个元素。严格大于 pivot 的部分也为空，j 指向数组的上界。

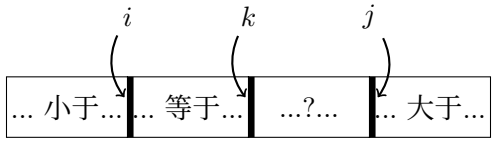


图 13.6: 基于 Lomuto 方法的三路划分

划分过程开始后，我们逐一检查 k 指向的元素。如果它等于 pivot，k 就移动指向下一个元素；如果它大于 pivot，我们将它和未处理区间的最后一个元素交换，这样严格大于的区间长度就增加一。它的边界 j 向左移动一步。由于我们不确定移动到 k 的元素是否仍然大于 pivot，我们需要再次进行比较，重复上述过程。否则，如果元素小于 pivot，我们将它和等于 pivot 区间的第一个元素交换。当 k 和 j 相遇时，划分过程结束。

```
1:  procedure SORT(A, l, u)
2:      if u − l > 1 then
3:          i ← l, j ← u, k ← l + 1
4:          pivot ← A[i]
5:          while k < j do
6:              while pivot < A[k] do
7:                  j ← j − 1
8:                  EXCHANGE A[k] ↔ A[j]
9:              if A[k] < pivot then
10:                 EXCHANGE A[k] ↔ A[i]
11:                 i ← i + 1
12:             k ← k + 1
13:         SORT(A, l, i)
14:         SORT(A, j, u)
```

和前面的三路划分快速排序算法相比，这一算法要相对简单，但是需要更多的交换次数。下面的 C 语言例子程序实现了这一算法。

```
void qsort(Key* xs, int l, int u) {
    int i, j, k; Key pivot;
    if (l < u - 1) {
        i = l; j = u; pivot = xs[l];
        for (k = l + 1; k < j; ++k) {
            while (pivot < xs[k]) { --j; swap(xs[j], xs[k]); }
            if (xs[k] < pivot) { swap(xs[i], xs[k]); ++i; }
        }
        qsort(xs, l, i);
        qsort(xs, j, u);
    }
}
```

练习 13.3

- 我们给出的命令式快速排序算法都使用第一个元素作为 pivot，也可以使用最后一个元素作为 pivot。请修改快速的排序的基本算法，Sedgewick 的改进算法，和三路快速排序算法，使用最后一个元素作为 pivot。

13.5 针对最差情况的工程实践

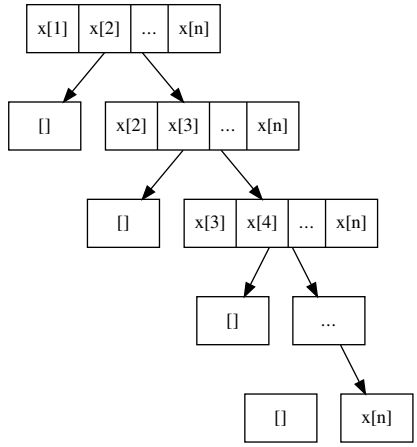
虽然三分快速排序（使用三路划分）能处理含有很多重复元素的序列，但是仍然无法有效解决典型的最差情况。例如，如果序列中的大部分元素已序时，无论是升序还是降序，划分的结果将会是两个长度不平衡的子序列，一个包含少量的元素，另一个包含剩余的部分。

考虑两种极端情况：$\{x_1 < x_2 < ... < x_n\}$ 和 $\{y_1 > y_2 > ... > y_n\}$。图13.7给出了划分结果。

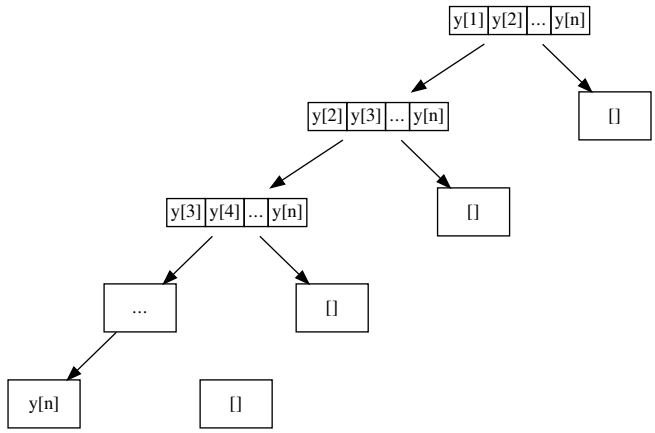
很容易给出更多的最差情况，例如 $\{x_m, x_{m-1}, ..., x_2, x_1, x_{m+1}, x_{m+2}, ...x_n\}$，其中 $\{x_1 < x_2 < ... < x_n\}$；另一个例子是 $\{x_n, x_1, x_{n-1}, x_2, ...\}$。图13.8给出了它们的划分结果。

观察可以发现，仅仅简单地选择第一个元素作为 pivot，很容易使得划分的结果不平衡，Robert Sedgwick 在 [69] 中给出了一种改进，在实际中得到了广泛的使用。这一改进不是每次在固定的位置上选择一个 pivot，而是进行简单的抽样以减小引发不平衡划分的可能性。一种抽样方法是检查第一个元素，中间的元素，和末尾的元素，然后选择这三个元素的中数（median）作为 pivot。在最差情况下，他保证划分后较短的序列至少含有一个元素。

在实际实现中还有一个细节需要注意。由于数组的索引在实际中的字长通常是有限的，简单使用 `(l + u) / 2` 来计算中间元素的索引可能引发溢出错误。正确的做法是使用 `l + (u - l) / 2` 来索引中间位置的元素。有两种方法来寻找中数，一种最多需要三次比较操作 [70]；另外一种方法通过交换将三个元素中的最小值移动到第一个元素的位置，将最大值移动到最后一个元素的位置，将中数移动到中间

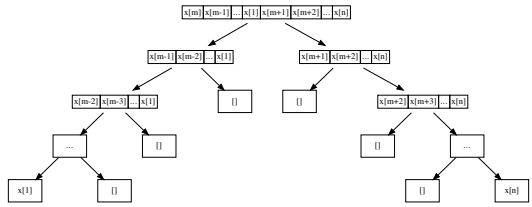


(a) 序列 $\{x_1 < x_2 < ... < x_n\}$ 的划分树，每次划分时，选择第一个元素为 pivot，小于等于 pivot 的部分总为空。

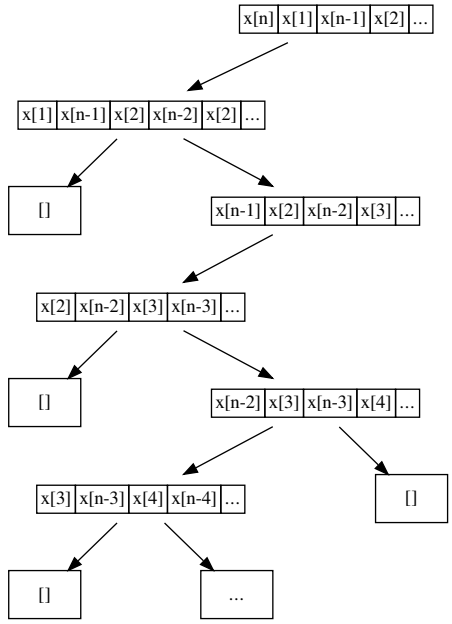


(b) 序列 $\{y_1 > y_2 > ... > y_n\}$ 的划分树，每次划分时，选择第一个元素为 pivot，大于等于 pivot 的部分总为空。

图 13.7: 两种最差情况



(a) 除了第一次划分结果，其他都不平衡。



(b) 一个 zig-zag 形状的划分树。

图 13.8: 另两种最差情况

位置。此后选在中间位置的元素作为 pivot 即可。下面的算法使用第二种方法确定划分的 pivot。

```
1: procedure SORT(A, l, u)
2:     if u − l > 1 then
3:         m ← ⌊(l+u)/2⌋                          ▷ 实际中要处理溢出的情况
4:         if A[m] < A[l] then                     ▷ 确保 A[l] ≤ A[m]
5:             EXCHANGE A[l] ↔ A[m]
6:         if A[u − 1] < A[l] then                 ▷ 确保 A[l] ≤ A[u − 1]
7:             EXCHANGE A[l] ↔ A[u − 1]
8:         if A[u − 1] < A[m] then                 ▷ 确保 A[m] ≤ A[u − 1]
9:             EXCHANGE A[m] ↔ A[u − 1]
10:        EXCHANGE A[l] ↔ A[m]
11:        (i, j) ← PARTITION(A, l, u)
12:        SORT(A, l, i)
13:        SORT(A, j, u)
```

对上述 4 种特殊的最差情况，这一算法显然性能良好。它常常被称为“三点中值”算法（median-of-three），我们将它的命令式实现留给读者作为练习。

但是，在纯函数式环境中，随机获取中间和最后的元素代价很大，我们不能直接将命令式的中数选择算法翻译过来。为了进行少量抽样，一种替代方案是在前三个元素中获取中数。如下面的 Haskell 例子程序所示。

```
qsort [] = []
qsort [x] = [x]
qsort [x, y] = [min x y, max x y]
qsort (x:y:z:rest) = qsort (filter (< m) (s:rest)) ++ [m] ++
                     qsort (filter (≥ m) (l:rest)) where
    xs = [x, y, z]
    [s, m, l] = [minimum xs, median xs, maximum xs]
```

但是，对于上述 4 种特殊的最差情况，这种替代方案都不能良好工作，本质原因是由于抽样的质量很差，我们需要在大范围内（整个列表），而不是在小范围内（前三个）进行抽样。我们稍后会介绍如果用函数式的方法解决这一划分问题。

除了 median-of-three 方法，另一种流行的工程实践是随机选择元素作为 pivot，例如下面的改进：

```
1: procedure SORT(A, l, u)
2:     if u − l > 1 then
3:         EXCHANGE A[l] ↔ A[ RANDOM(l, u) ]
4:         (i, j) ← PARTITION(A, l, u)
5:         SORT(A, l, i)
6:         SORT(A, j, u)
```

函数 RANDOM$(l, u$ 返回一个在 l 和 u 之间的随机整数 $l \leq i < u$。这一位置上的元素被交换到第一位置上作为 pivot 用以进行划分。这一算法称为随机快速排序[4]。

理论上，无论 median-of-three 还是随机快速排序都不能完全避免最差情况。如果待排序序列是随机分布的，无论选择第一个作为 pivot，还是任何其他位置上的元素，在效果上都是相同的。在纯函数式编程环境中，列表的底层数据结构通常是单向链表，没有简单的方法可以实现纯函数式的随机快速排序。

即使在理论上无法避免最差情况，但是这些工程上的实践在实际应用中往往能够取得很好的结果。

13.6 其他工程实践

还有一些工程实践，它们不是着眼于解决划分的最差情况。Robert Sedgewick 观察到如果待排序的列表较短时，快速排序引入的额外代价比较明显，此时插入排序反而更有优势 [2]、[70]。Sedgewick、Bentley 和 McIlroy 尝试了不同的序列长度，称为“Cut-Off”。如果序列中的元素个数少于 Cut-Off，就转而使用插入排序。

```
1: procedure SORT(A, l, u)
2:     if u − l > CUT-OFF then
3:         QUICK-SORT(A, l, u)
4:     else
5:         INSERTION-SORT(A, l, u)
```

这一改进的实现留给读者作为练习。

练习 13.4

- 除了本节给出的 4 种最差情况外，请给出更多的最差情况。
- 选择一门命令式编程语言，实现 median-of-three 方法。
- 选择一门命令式编程语言，实现随机快速排序。
- 使用命令式方法和函数式方法，实现当列表长度较短时改用插入排序的算法。

13.7 其他

有人说只有应用了全部改进技术的实现才是“真正的快速排序”——当序列较短时转而使用插入排序，并且就地交换元素，同时用 median-of-tree 选择 pivot，再加上三路划分。最简短的纯函数式实现，虽然完美地诠释了快速排序的思路，却没有使用上述任何改进技术。有人认为纯函数式的快速排序本质上是树排序。

事实上,快速排序和树排序有紧密的关系。Richard Bird 展示了通过 deforestation,从二叉树排序推导出快速排序 [72]。

考虑一个生成二叉搜索树的算法，名为 $unfold$。它将一个元素列表转换为一棵二叉搜索树。

$$unfold(L) = \begin{cases} \phi & : & L = \phi \\ tree(T_l, l_1, T_r) & : & otherwise \end{cases} \tag{13.25}$$

其中

$$\begin{aligned} T_l &= unfold(\{a | a \in L', a \leq l_1\}) \\ T_r &= unfold(\{a | a \in L', l_1 < a\}) \end{aligned} \tag{13.26}$$

有趣的一点是，和此前二叉搜索树一章介绍的内容相比，这一算法产生树的方式大相径庭。如果要进行 unfold 的列表为空，结果显然为一棵空树。这是边界条件。否则，算法将列表中第一个元素 l_1 作为节点的 key，然后递归地创建左右子树。用于创建左子树的元素，是列表 L' 中小于等于 key 的元素；而其他大于 key 的元素被用以创建右子树。其中 L' 是 L 中除 l_1 外的剩余部分。

此前我们给出过将一棵二叉搜索树通过中序遍历转换成列表的算法:

$$toList(T) = \begin{cases} \phi & : & T = \phi \\ toList(left(T)) \cup \{key(T)\} \cup toList(right(T)) & : & otherwise \end{cases} \tag{13.27}$$

我们可以将上述两个函数组合（compose）起来，定义出快速排序算法:

$$quickSort = toList \cdot unfold \tag{13.28}$$

第一步，我们通过 $unfold$ 构造一棵二叉搜索树。将其作为中间结果送入 $toList$ 得出列表后就可以将这棵树丢弃了。如果将这一临时的中间结果树消除，就得到了基本的快速排序算法。

消除临时的中间结果二叉搜索树的过程称作deforestation。这一概念来自 Burstle-Darlington 的工作 [73]。

13.8 归并排序

虽然快速排序在大多数情况下表现出众，但是在最坏情况下性能无法得到保证。即使各种工程上实践上的改进，也无法完全避免最坏情况。归并排序，能够在所有情况下都保证 $O(n \lg n)$ 的性能。在算法的理论设计和分析上特别重要。此外，归并排序特别适于空间上链接的场景，可以对非连续存储的序列进行的排序。某些函数式编程环境和动态编程环境，往往使用归并排序作为标准库中的排序方案，包括 Haskell、Python 和 Java（Java 7 之后）。

本节中，我们首先介绍归并排序的直观思想，给出基本实现。然后，我们介绍一些归并排序的变形，包括自然归并排序和自底向上的归并排序。

13.8.1 基本归并排序

和快速排序一样，归并排序本质上也是使用分而治之的策略。和快速排序不同，归并排序保证划分是严格平衡的，它每次都将待排序序列从中间位置分割开。然后它递归地对子序列排序，并将两个子序列的排序结果归并。算法可以描述如下。

当对序列 L 排序时，

- 边界情况：如果序列为空，则结果显然也为空；
- 否则，将序列从中间位置分成两部分，递归对两个子序列排序，然后将结果归并。

基本归并排序算法可以形式化为下面的公式。

$$sort(L) = \begin{cases} \phi & : \quad L = \phi \\ merge(sort(L_1), sort(L_2)) & : \quad L \neq \phi, (L_1, L_2) = splitAt(\lfloor \frac{|L|}{2} \rfloor, L) \end{cases} \tag{13.29}$$

归并

上面的归并排序定义中，有两个“黑盒子”。一个是 $splitAt$ 函数，它从指定的位置将序列分割成两部分；另外一个是 $merge$ 函数，它可以将两个已序序列合成一个。

如本书附录所示，在命令式环境中，由于可以使用随机索引，实现 $splitAt$ 非常简单。但是在函数式环境中，它通常实现为一个线性时间的算法：

$$splitAt(n, L) = \begin{cases} (\phi, L) & : \quad n = 0 \\ (\{l_1\} \cup A, B) & : \quad n \neq 0, (A, B) = splitAt(n-1, L') \end{cases} \tag{13.30}$$

其中 l_1 是非空列表 L 的第一个元素，L' 包含除 l_1 之外的剩余部分。

归并的思想如图13.9所示。考虑两队小孩，他们已经按照身高的顺序站好队。最矮的孩子在前面，最高的孩子在后面。

现在，我们要求这些孩子依次通过一扇门，每次只能有一个小孩通过。并且必须按照身高的顺序。任何一个孩子，只有所有比他矮的其他小孩通过后，才能通过这扇门。

由于两队小孩都“已序”了，我们可以让每队最前面的两个孩子互相比较身高，较矮的一个孩子可以通过门；然后我们重复这一步骤，直到任何一队的小孩都已经通过门了，此后剩下的一队中的孩子们可以逐一通过这扇门。

下面的公式描述了这一思路。

$$merge(A, B) = \begin{cases} A & : \quad B = \phi \\ B & : \quad A = \phi \\ \{a_1\} \cup merge(A', B) & : \quad a_1 \leq b_1 \\ \{b_1\} \cup merge(A, B') & : \quad otherwise \end{cases} \tag{13.31}$$

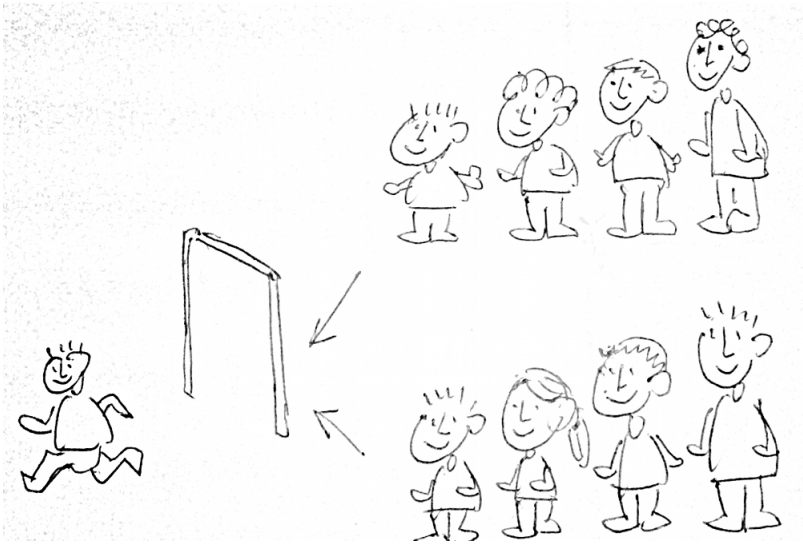


图 13.9: 两队孩子通过一扇门

其中 a_1 和 b_1 分别是列表 A 和 B 中的第一个元素；A' 和 B' 分别是出第一元素外的剩余部分。式中的前两种情况是简单的边界情况：将一个已序列表和一个空列表归并的结果就是这一列表本身；否则，如果两个列表都不为空，我们从两个列表中各自取出第一个元素，将它们进行比较，取较小的作为结果中的第一个元素，然后递归对剩余的部分进行归并。

下面的 Haskell 例子程序，使用 $merge$ 的定义，实现了完整的归并排序。

```
msort [] = []
msort [x] = [x]
msort xs = merge (msort as) (msort bs) where
  (as, bs) = splitAt (length xs `div` 2) xs

merge xs [] = xs
merge [] ys = ys
merge (x:xs) (y:ys) | x ≤ y = x : merge xs (y:ys)
                    | x > y = y : merge (x:xs) ys
```

注意，这一实现和上面的算法定义略有不同，它将只含有一个元素的情况也算作边界情况处理。

归并排序也可以用命令式的方式实现，下面给出了基本的归并排序算法。

```
1: procedure SORT(A)
2:     if |A| > 1 then
3:         m ← ⌊|A|/2⌋
4:         X ← COPY-ARRAY(A[1...m])
5:         Y ← COPY-ARRAY(A[m + 1...|A|])
6:         SORT(X)
7:         SORT(Y)
```

8: MERGE(A, X, Y)

当待排序数组包含至少两个元素时，开始进行处理。首先将前一半元素复制到一个新数组 X 中，将后一半复制到数组 Y 中。然后递归对它们排序，最后将排序结果归并回 A 中。这一方法使用了和 A 大小相同的额外空间。这是由于 MERGE 算法不是在原地修改元素的。我们稍后将介绍命令式的原地归并排序算法。

归并过程所做的处理和此前给出的函数式定义相同。存在一个较复杂的实现，和一个使用 sentinel 的简化实现。

较复杂的归并算法不断检查两个输入数组的元素，选择较小的一个并放回结果数组 A，它接着继续向前处理直到任何一个数组被处理完。此后算法将另一个数组中的剩余元素添加到 A。

1: **procedure** MERGE(A, X, Y)
2: $i \leftarrow 1, j \leftarrow 1, k \leftarrow 1$
3: $m \leftarrow |X|, n \leftarrow |Y|$
4: **while** $i \leq m \wedge j \leq n$ **do**
5: **if** $X[i] < Y[j]$ **then**
6: $A[k] \leftarrow X[i]$
7: $i \leftarrow i + 1$
8: **else**
9: $A[k] \leftarrow Y[j]$
10: $j \leftarrow j + 1$
11: $k \leftarrow k + 1$
12: **while** $i \leq m$ **do**
13: $A[k] \leftarrow X[i]$
14: $k \leftarrow k + 1$
15: $i \leftarrow i + 1$
16: **while** $j \leq n$ **do**
17: $A[k] \leftarrow Y[j]$
18: $k \leftarrow k + 1$
19: $j \leftarrow j + 1$

虽然这一算法较为繁复，但在某些具有丰富数组处理工具的编程环境中，也可以获得简洁的实现。如下面的 Python 例子程序所示。

```
def msort(xs):
    n = len(xs)
    if n > 1:
        ys = [x for x in xs[:n/2]]
        zs = [x for x in xs[n/2:]]
        ys = msort(ys)
        zs = msort(zs)
        xs = merge(xs, ys, zs)
```

```
    return xs

def merge(xs, ys, zs):
    i = 0
    while ys ≠ [] and zs ≠ []:
        xs[i] = ys.pop(0) if ys[0] < zs[0] else zs.pop(0)
        i = i + 1
    xs[i:] = ys if ys ≠[] else zs
    return xs
```

性能

在对基本归并排序进行改进前，我们先分析一下归并排序的性能。算法分为两步：分解步骤和归并步骤。在分解步骤中，待排序序列总是被分成两个长度相等的子序列。如果我们仿照快速排序的方式画一棵划分树，可以得到一棵完美平衡的二叉树，如图13.3所示。因此这棵树的高度为 $O(\lg n)$。也就是说归并排序的递归深度为 $O(\lg n)$。在递归的每一层，都会发生归并操作。归并算法的性能分析很直观，他总是成对比较输入序列的元素，当其中一个序列被处理完后，另一个序列中的元素被逐一复制到结果中，因此它是一个线性时间算法，复杂度和序列的长度成比例。根据这一事实，记 $T(n)$ 为对长度为 n 的序列进行排序所需要的时间，我们可以写出递归的时间开销如下：

$$\begin{aligned} T(n) &= T(\frac{n}{2}) + T(\frac{n}{2}) + cn \\ &= 2T(\frac{n}{2}) + cn \end{aligned} \tag{13.32}$$

排序的时间包含三部分：对前半部分进行归并排序耗时 $T(\frac{n}{2})$，对后半部分归并排序也耗时 $T(\frac{n}{2})$，将两部分结果归并用时 cn，其中 c 是某个常数。解此方程得到结果为 $O(n \lg n)$。

这一性能结果对所有情况都适用，这是因为归并排序总是将输入序列平均分成两部分。

另外一个重要的性能指标是空间消耗。但是不同的归并排序实现方法，空间消耗大相径庭。我们稍后介绍每一种具体实现时，会对空间复杂度进行详细的分析。

对于前面给出的最基本的归并排序实现，在每一次递归时，都需要和输入数组同样大小的空间，用以复制元素和进一步的递归排序，这一层的递归返回后，这些空间可以释放。因此最大的空间消耗出现在进入最深一层递归时，为 $O(n \lg n)$。

函数式归并排序消耗的空间远远小于这一结果，这是因为序列底层的数据结构为链表。我们无需额外的空间以进行归并[4]。最主要的空间消耗来自于递归调用栈。稍后介绍奇偶分割算法时，我们会再次解释空间消耗的问题。

[4]我们这里忽略惰性求值引入的更复杂的因素，可以参考 [72] 了解详细的分析。

细微改进

我们接下来将逐步改进函数式和命令式的归并排序算法。前面给出的命令式归并算法比较冗长。我们可以使用正无穷作为 sentinel 来简化 [4]。我们将 ∞ 添加到两个待归并的已序数组的末尾[5]。这样就无需检查数组是否已用完。图13.10描述了这一思路。

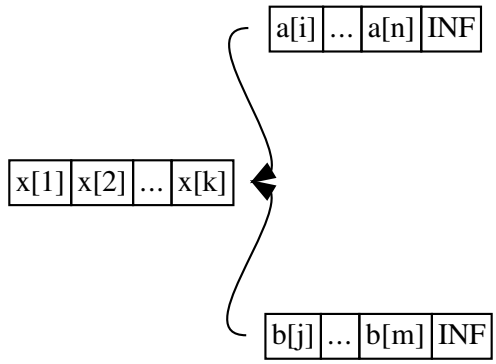


图 13.10: 使用 ∞ 作为 sentinels 来简化归并

```
1: procedure MERGE(A, X, Y)
2:     APPEND(X, ∞)
3:     APPEND(Y, ∞)
4:     i ← 1, j ← 1
5:     for k ← from 1 to |A| do
6:         if X[i] < Y[j] then
7:             A[k] ← X[i]
8:             i ← i + 1
9:         else
10:            A[k] ← Y[j]
11:            j ← j + 1
```

下面的 C 语言例子程序实现了这一简化。它将归并算法嵌入到了排序中。`INF` 被定义为一个大常数，类型和 `Key` 一致。类型可以预先定义，或者将类型信息通过一个比较函数进行抽象，在将比较函数作为一个参数传入排序算法中。我们在此忽略这些语言细节。

```
void msort(Key* xs, int l, int u) {
    int i, j, m;
    Key *as, *bs;
    if (u - l > 1) {
        m = l + (u - l) / 2;  //防止int溢出
```

[5]如果是按照单调非递增顺序排序，则使用 $-\infty$

```
        msort(xs, l, m);
        msort(xs, m, u);
        as = (Key*) malloc(sizeof(Key) * (m - l + 1));
        bs = (Key*) malloc(sizeof(Key) * (u - m + 1));
        memcpy((void*)as, (void*)(xs + l), sizeof(Key) * (m - l));
        memcpy((void*)bs, (void*)(xs + m), sizeof(Key) * (u - m));
        as[m - l] = bs[u - m] = INF;
        for (i = j = 0; l < u; ++l)
            xs[l] = as[i] < bs[j] ? as[i++] : bs[j++];
        free(as);
        free(bs);
    }
}
```

运行这一程序所需的时间远远超过快速排序。除了稍后会介绍的最主要原因外，在归并时反复申请和释放内存也是一个需要改进的地方。内存申请是实际应用程序中的一个常见瓶颈 [2]。一个解决方法是一次性申请一个和待排序数组同样大小的空间作为工作区（working area）。此后，对前、后两半部分的递归排序就无需申请额外的空间，而是用工作区来进行归并。最后算法再将工作区内的结果复制回原数组。

下面的算法实现了这一改进的归并排序。

```
1: procedure SORT(A)
2:     B ← CREATE-ARRAY(|A|)
3:     SORT′(A, B, 1, |A|)

4: procedure SORT′(A, B, l, u)
5:     if u − l > 0 then
6:         m ← ⌊(l+u)/2⌋
7:         SORT′(A, B, l, m)
8:         SORT′(A, B, m + 1, u)
9:         MERGE′(A, B, l, m, u)
```

这一算法创建了另一个同样大小的数组，并将其作为一个参数和原待排序数组一同传入 SORT′ 算法。在实际的实现中，这一工作区最终需要人工释放，或者使用自动工具如 GC（垃圾回收）释放。修改后的归并算法 MERGE′ 也接受一个工作区参数。

```
1: procedure MERGE′(A, B, l, m, u)
2:     i ← l, j ← m + 1, k ← l
3:     while i ≤ m ∧ j ≤ u do
4:         if A[i] < A[j] then
5:             B[k] ← A[i]
6:             i ← i + 1
7:         else
8:             B[k] ← A[j]
```

```
        j ← j + 1
    k ← k + 1
while i ≤ m do
    B[k] ← A[i]
    k ← k + 1
    i ← i + 1
while j ≤ u do
    B[k] ← A[j]
    k ← k + 1
    j ← j + 1
for i ← from l to u do                    ▷ 复制回
    A[i] ← B[i]
```

通过这一小改进，归并排序所需要的空间从 $O(n \lg n)$ 降低到 $O(n)$。下面的 C 语言例子程序实现了这一改进。出于示例的目的，我们在一个循环中逐一将归并结果复制会原数组。在实际中通常使用标准库中提供的工具，如 `memcpy`。

```
void merge(Key* xs, Key* ys, int l, int m, int u) {
    int i, j, k;
    i = k = l; j = m;
    while (i < m && j < u)
        ys[k++] = xs[i] < xs[j] ? xs[i++] : xs[j++];
    while (i < m)
        ys[k++] = xs[i++];
    while (j < u)
        ys[k++] = xs[j++];
    for(; l < u; ++l)
        xs[l] = ys[l];
}

void msort(Key* xs, Key* ys, int l, int u) {
    int m;
    if (u - l > 1) {
        m = l + (u - l) / 2;
        msort(xs, ys, l, m);
        msort(xs, ys, m, u);
        merge(xs, ys, l, m, u);
    }
}

void sort(Key* xs, int l, int u) {
    Key* ys = (Key*) malloc(sizeof(Key) * (u - l));
    kmsort(xs, ys, l, u);
    free(ys);
}
```

改进后的程序运行速度明显加快。在我的测试计算机上，对 100000 个随机产生的元素排序时，速度能够提升 20% 到 25%。

函数式归并排序也可以进一步改进。前面给出的版本在列表中间位置将其分成两部分。但是，由于列表本质上是单向链表，对给定位置进行随机访问是一个线性时间的操作（详细信息可以参考附录 A）。作为改进，我们可以使用奇偶位置分割列表。这样所有位于奇数位置的元素被放入一个子列表，而所有偶数位置的元素被放入另一个子列表。对于任意列表，奇偶位置的元素要么同样多，要么仅相差一个。因此这一分割策略总能保证平衡分割，总性能在任何情况下都为 $O(n \lg n)$。

奇偶分割算法可以定义如下：

$$split(L) = \begin{cases} (\phi, \phi) & : \quad L = \phi \\ (\{l_1\}, \phi) & : \quad |L| = 1 \\ (\{l_1\} \cup A, \{l_2\} \cup B) & : \quad otherwise, (A, B) = split(L'') \end{cases} \tag{13.33}$$

如果列表为空，分割的结果为两个空列表；如果列表仅含有一个元素，我们将此位置为 1 的元素放入奇数位置子列表中，而偶数位置子列表为空；否则，列表中至少含有两个元素，我们将第一个元素放入奇数位置子列表，将第二个元素放入偶数位置子列表，然后递归对剩余元素进行奇偶分割。

剩余的函数保持不变，下面的 Haskell 例子程序给出了奇偶分割算法的实现。

```
split [] = ([], [])
split [x] = ([x], [])
split (x:y:xs) = (x:xs', y:ys') where (xs', ys') = split xs
```

13.9 原地归并排序

命令式归并排序的一个主要缺点是需要额外的空间以进行归并，不带优化的基本实现在高峰时需要 $O(n \lg n)$ 的空间，使用工作区优化后也仍然需要 $O(n)$ 的空间。

这使得人们去探索原地归并排序，通过复用原待排序数组而不申请额外空间。本节中，我们将介绍实现原地归并排序的一些解法。

13.9.1 死板的原地归并

第一个想法很直观。如图13.11所示，子数组 A 和 B 已排序好，当进行原地归并时，我们规定一个不变性质，令 i 之前的所有元素为已归并完成的部分，它们满足非递减的顺序；每次比较第 i 个元素和第 j 个元素。如果第 i 个元素小于第 j 个元素，就将 i 向前移动一步。这种情况比较简单；否则，说明第 j 个元素应该放入下一个归并结果中，位置在 i 之前。为了达到这一点，所有 i 和 j 之间的元素，包括第 i 个元素，都要向后移动一个位置。我们重复这一步骤，直到所有 A 和 B 中的元素都置于正确的位置。

1: **procedure** MERGE(A, l, m, u)
2: **while** $l \leq m \wedge m \leq u$ **do**
3: **if** $A[l] < A[m]$ **then**

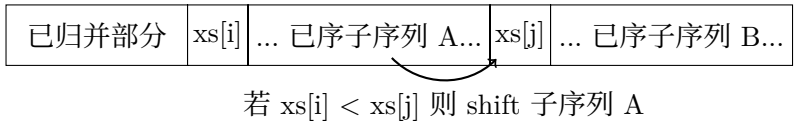


图 13.11: 死板原地归并

```
4:          l ← l + 1
5:      else
6:          x ← A[m]
7:          for i ← m down-to l + 1 do                    ▷ Shift
8:              A[i] ← A[i − 1]
9:          A[l] ← x
```

但是，这一死板的解法使得归并排序的性能退化为平方级 $O(n^2)$。这是因为数组的移动是一个线性时间的操作，它和第一个子数组中尚未归并的元素个数成正比。

依照这一方法实现的的 C 语言例子程序运行速度很慢，对 10000 个随机生成的元素排序时，它消耗的时间比前面给出的程序多 12 倍。

```
void naive_merge(Key* xs, int l, int m, int u) {
    int i; Key y;
    for(; l < m && m < u; ++l)
        if (!(xs[l] < xs[m])) {
            y = xs[m++];
            for (i = m - 1; i > l; --i) /* shift */
                xs[i] = xs[i-1];
            xs[l] = y;
        }
}

void msort3(Key* xs, int l, int u) {
    int m;
    if (u - l > 1) {
        m = l + (u - l) / 2;
        msort3(xs, l, m);
        msort3(xs, m, u);
        naive_merge(xs, l, m, u);
    }
}
```

13.9.2 原地工作区

为了能在 $O(n \lg n)$ 时间内实现原地归并排序，当对子数组排序时，必须使用数组剩余的部分作为归并的工作区。对于已经在工作区内的元素，由于稍后也要进行排序，它们不能被覆盖。我们可以修改此前申请同样大小额外空间的程序来实现这一点。思路如下：当我们比较两个已序的子数组的最前面的元素时，如果要将较小的元

素放入工作区中的某个位置，我们同时将工作区中的这个元素和选出的较小的元素交换。这样，当归并完成后，原来的两个子数组就保存了此前工作区中存储的内容。如图13.12所示。

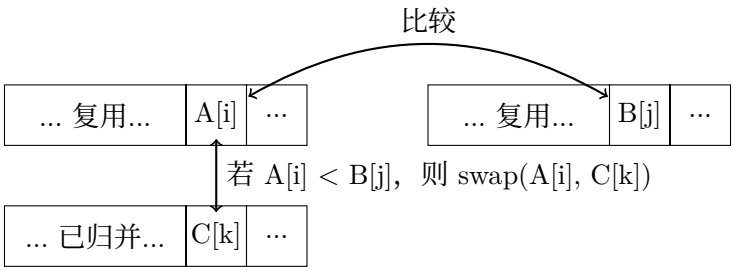


图 13.12: 归并时不覆盖工作区中的内容

在改进的算法中，两个已序的子数组，和用于归并的工作区都是最初的待排序数组中的一部分。归并时需要提供的参数包括：两个已序数组的起始和结束位置，可以用区间来表示它们；另外还需要提供工作区的起始位置。下面的算法使用 $[a, b)$ 来表示左闭右开区间，它包括 a，但不包括 b。算法将已序区间 $[i, m)$ 和 $[j, n)$ 归并到从 k 开始的工作区。

```
1:  procedure MERGE(A, [i, m), [j, n), k)
2:      while i < m ∧ j < n do
3:          if A[i] < A[j] then
4:              EXCHANGE A[k] ↔ A[i]
5:              i ← i + 1
6:          else
7:              EXCHANGE A[k] ↔ A[j]
8:              j ← j + 1
9:          k ← k + 1
10:     while i < m do
11:         EXCHANGE A[k] ↔ A[i]
12:         i ← i + 1
13:         k ← k + 1
14:     while j < m do
15:         EXCHANGE A[k] ↔ A[j]
16:         j ← j + 1
17:         k ← k + 1
```

注意，在归并时必须满足下面的两个限制条件：

1. 工作区必须在数组的边界内。也就是说，工作区必须足够大，以容纳交换进来的元素而不会引起越界错误；

2. 工作区可以和任何一个已序的子数组存在重叠，但是必须保证尚未归并的元素不会被覆盖。

下面的 C 语言例子程序实现了这一算法。

```
void wmerge(Key* xs, int i, int m, int j, int n, int w) {
    while (i < m && j < n)
        swap(xs, w++, xs[i] < xs[j] ? i++ : j++);
    while (i < m)
        swap(xs, w++, i++);
    while (j < n)
        swap(xs, w++, j++);
}
```

使用这一算法，我们很容易想出一个解法，能够将数组的一半内容进行归并排序。接下来的问题是，如何处理剩下的一半尚未排序的元素？如图13.13所示。

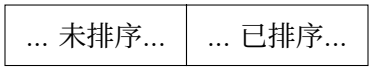


图 13.13: 数组中的一半被排序

一个直观的想法是递归对工作区中的一半内容进行排序，这样就只剩下 $\frac{1}{4}$ 的元素尚未排序了。结果如图13.14所示。这里关键的一点是，我们必须在某个时候将已序的 $\frac{1}{4}$ 元素 B 和已序的 $\frac{1}{2}$ 元素 A 归并。

图 13.14: A 和 B 必须在某个时刻归并到一起

但是，剩余的工作区，其大小可以容纳 $\frac{1}{4}$ 元素，它足够容纳 A 和 B 的归并结果么？不幸的是，在如图13.14所示的布局中，这一空间是不够用的。

但是，上述的第二条限制条件启发我们：能否通过某种归并的设计，保证未归并的元素不被覆盖，从而利用工作区和已序子数组的重叠部分来解决这个问题？

实际上，我们可以先不让工作区的后二分之一元素已序，而让前二分之一部分已序，这样工作区就位于两段已序子数组的中间，如图13.15 (a) 所示。这样的安排就使得工作区和子数组 A 产生了重叠 [74]。

考虑两种极端情况：

1. 所有 B 中的元素都小于 A 中的任意元素。这种情况下，归并算法最终将 B 中的全部内容移动到工作区中；而 B 中将包括以前工作区中所保存的内容；由于工作区和 B 的大小相等，因此恰好可以交换它们的内容；

2. 所有 A 中的元素都小于 B 中的任意元素。这种情况下，归并算法不断交换 A 和工作区中的元素。当工作区的前 $\frac{1}{4}$ 区间被 A 中的元素填满后，算法开始覆盖

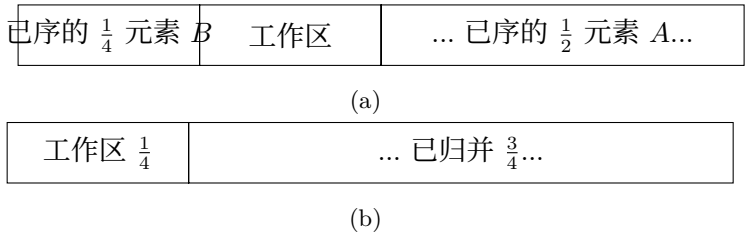


图 13.15: 利用工作区归并子数组 A 和 B

A 的前一半部分的内容。幸运的是，被覆盖的内容不是未归并的元素。工作区的边界不断向数组的末尾移动，并最终达到最右侧；此后，归并算法开始交换 B 和工作区的内容。最终工作区被移动到了数组的最左侧，如图13.15 (b) 所示。

我们可以重复这一步骤，总是对未排序部分的后二分之一排序，从而将已序结果交换到前一半，而使得新的工作区位于中间。这样就不断将工作区的大小减半，从 $\frac{1}{2}$ 到 $\frac{1}{4}$ 到 $\frac{1}{8}$……归并的规模不断下降。当工作区中只剩下一个元素时，我们无须继续排序，因为只含有一个元素的数组自然是已序的。归并只含有一个元素的数组等价于插入元素。实际上，我们可以使用插入排序来处理最后的几个元素。

完整的算法可以描述如下：

```
1:  procedure SORT(A, l, u)
2:      if u − l > 0 then
3:          m ← ⌊(l+u)/2⌋
4:          w ← l + u − m
5:          SORT'(A, l, m, w)                          ▷ 后半部分包含已序元素
6:          while w − l > 1 do
7:              u' ← w
8:              w ← ⌈(l+u')/2⌉                         ▷ 保证工作区足够大
9:              SORT'(A, w, u', l)                     ▷ 前半部分包含已序元素
10:             MERGE(A, [l, l + u' − w], [u', u], w)
11:         for i ← w down-to l do                     ▷ 改用插入排序
12:             j ← i
13:             while j ≤ u ∧ A[j] < A[j − 1] do
14:                 EXCHANGE A[j] ↔ A[j − 1]
15:                 j ← j + 1
```

为了满足第一个限制条件，我们必须保证工作区足够大以容纳全部交换进来的元素，因此在对后一半排序时，我们总是使用上限取整。我们将包含结束位置的区间信息传入了 MERGE 算法。

接下来，我们需要定义Sort'算法，它反过来递归调用Sort来交换工作区和已序部分。

```
1: procedure SORT'(A, l, u, w)
2:     if u − l > 0 then
3:         m ← ⌊(l+u)/2⌋
4:         SORT(A, l, m)
5:         SORT(A, m + 1, u)
6:         MERGE(A, [l, m], [m + 1, u], w)
7:     else                                    ▷ 将所有元素交换到工作区
8:         while l ≤ u do
9:             EXCHANGE A[l] ↔ A[w]
10:            l ← l + 1
11:            w ← w + 1
```

和前面的死板原地归并排序不同，这一方法在归并中并不 shift 元素。未排序部分的长度不断递减：$\frac{n}{2}, \frac{n}{4}, \frac{n}{8}, ...$，总共需要 $O(\lg n)$ 步完成排序。每次递归对剩余部分的一半排序，然后使用线性时间进行归并。

记对 n 个元素排序所花费的时间为 $T(n)$，我们有如下的等式：

$$T(n) = T(\frac{n}{2}) + c\frac{n}{2} + T(\frac{n}{4}) + c\frac{3n}{4} + T(\frac{n}{8}) + c\frac{7n}{8} + ... \tag{13.34}$$

对于一半的元素，花费的时间为：

$$T(\frac{n}{2}) = T(\frac{n}{4}) + c\frac{n}{4} + T(\frac{n}{8}) + c\frac{3n}{8} + T(\frac{n}{16}) + c\frac{7n}{16} + ... \tag{13.35}$$

两式相减 (13.34) - (13.35) 得：

$$T(n) - T(\frac{n}{2}) = T(\frac{n}{2}) + cn(\frac{1}{2} + \frac{1}{2} + ...)$$

共有 $\lg n$ 个 $\frac{1}{2}$ 相加，由此得到计算时间的递归关系为：

$$T(n) = 2T(\frac{1}{2}) + cn\lg n$$

使用裂项求和（telescoping）方法解此方程，可以得到结果 $O(n\lg^2 n)$。

下面的 C 语言例子程序给出了这一算法的完整实现,它使用了前面给出的 `wmerge` 函数。

```
void imsort(Key* xs, int l, int u);

void wsort(Key* xs, int l, int u, int w) {
    int m;
    if (u - l > 1) {
        m = l + (u - l) / 2;
        imsort(xs, l, m);
        imsort(xs, m, u);
        wmerge(xs, l, m, m, u, w);
    }
```

```
    else
        while (l < u)
            swap(xs, l++, w++);
}

void imsort(Key* xs, int l, int u) {
    int m, n, w;
    if (u - l > 1) {
        m = l + (u - l) / 2;
        w = l + u - m;
        wsort(xs, l, m, w);  //后半部分包含了已序元素。
        while (w - l > 2) {
            n = w;
            w = l + (n - l + 1) / 2; //向上取整
            wsort(xs, w, n, l);  //前半部分包含已序元素。
            wmerge(xs, l, l + n - w, n, u, w);
        }
        for (n = w; n > l; --n) //切换到插入排序
            for (m = n; m < u && xs[m] < xs[m-1]; ++m)
                swap(xs, m, m - 1);
    }
}
```

但是，和前面给出的预先分配同等大小的数组用于归并的程序相比，这一程序的运行速度并不快。在我的测试计算机上，对 100000 个随机产生的元素排序时，它的运行速度要慢 60%，这主要是由于大量的交换操作造成的。

13.9.3 原地归并排序 vs. 链表归并排序

原地归并排序仍然是一个活跃的研究领域。减少归并所需的额外空间是有代价的，它增加了归并排序算法的复杂程度。但是，如果待排序的序列不是存储在数组中，而是用链表来表示，归并就无需额外的空间。如前面的奇偶归并排序算法所示。

为了对比，我们可以给出一个纯命令式的链表归并排序实现。链表节点可以定义为一个结构，如下面的 C 语言例子所示：

```
struct Node {
    Key key;
    struct Node* next;
};
```

我们可以定义一个辅助函数用于节点连接。设待连接的链表不为空，下面的 C 语言例子程序实现了连接函数。

```
struct Node* link(struct Node* x, struct Node* ys) {
    x→next = ys;
    return x;
}
```

为了实现命令式的奇偶分割，我们初始化两个空的子列表。然后遍历待分割的列表。每次迭代，我们将当前的节点连接到第一个子列表的前面，然后交换两个子列表，

这样下次迭代时，节点就会连接到第二个子列表的前面。这一方法可以描述如下：

```
1: function SPLIT(L)
2:     (A, B) ← (φ, φ)
3:     while L ≠ φ do
4:         p ← L
5:         L ← NEXT(L)
6:         A ← LINK(p, A)
7:         EXCHANGE A ↔ B
8:     return (A, B)
```

下面的 C 语言例子程序实现了这一分割算法，并将其嵌入到排序函数中。

```
struct Node* msort(struct Node* xs) {
    struct Node *p, *as, *bs;
    if (!xs || !xs→next) return xs;

    as = bs = NULL;
    while(xs) {
        p = xs;
        xs = xs→next;
        as = link(p, as);
        swap(as, bs);
    }
    as = msort(as);
    bs = msort(bs);
    return merge(as, bs);
}
```

接下来需要实现链表的命令式归并算法。思路和数组的归并类似。不断比较两个列表的第一个元素，选择较小的附加到结果列表的末尾。当任一列表变空时，将另外一个列表连接到结果的后面，而无需逐一复制。结果列表在初始化时需要额外的判断，这是因为表头要指向两个列表中首元素较小的一个。一种简化处理是使用一个 dummy 的 sentinel 的表头，最后在返回结果前将它去掉。下面的例子程序给出了详细的实现。

```
struct Node* merge(struct Node* as, struct Node* bs) {
    struct Node s, *p;
    p = &s;
    while (as && bs) {
        if (as→key < bs→key) {
            link(p, as);
            as = as→next;
        }
        else {
            link(p, bs);
            bs = bs→next;
        }
        p = p→next;
    }
```

```
    if (as)
        link(p, as);
    if (bs)
        link(p, bs);
    return s.next;
}
```

练习 13.5

- 证明原地归并排序的性能为 $O(n \lg n)$。

13.10 自然归并排序

Knuth 给出了另外一种方法来实现分而治之的归并排序。整个过程如同从两端点燃一支蜡烛 [51]，称为自然归并排序算法。

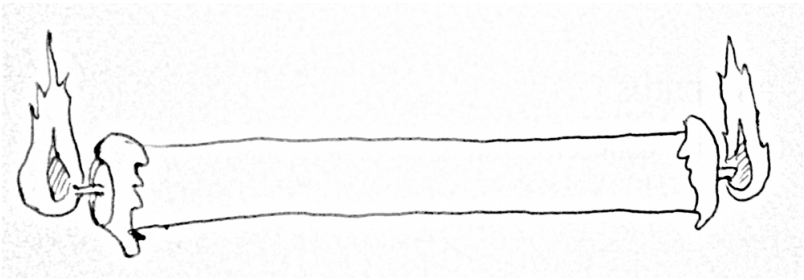


图 13.16: 从两端向中间燃烧的蜡烛

对于任何序列，可以在任何位置开始找到一个非递减子序列。作为一个特殊情况，我们总可以在最左侧找到这样的子序列。下表给出了一些例子，非递减子序列用下划线标出。

15 , 0, 4, 3, 5, 2, 7, 1, 12, 14, 13, 8, 9, 6, 10, 11
8, 12, 14, 0, 1, 4, 11, 2, 3, 5, 9, 13, 10, 6, 15, 7
0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15

表 13.3: 非递减子序列的例子

表中的第一行描述了最差的情况，第二个元素小于第一个，因此非递减子序列长度为一，只包含第一个元素；表中的最后一行描述了最好的情况，整个序列已序，非递减子序列包含全部元素；表中的第二行描述了通常的情况。

对称地，我们同样总是可以从序列的右端向左找到一个非递减子序列。于是，我们可以将两个非递减子序列，一个从头部开始，一个从尾部开始，归并成一个更长的

序列。这一思路的最大优点是，我们可以利用子序列元素间的自然顺序，而无需递归排序。

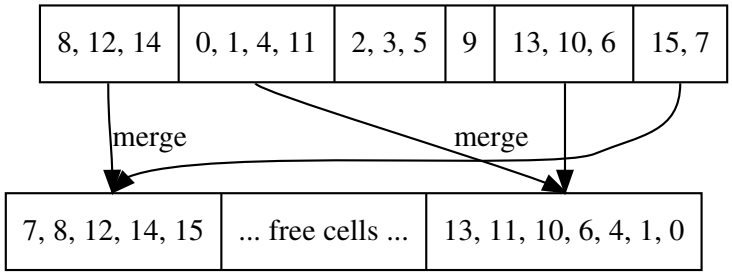


图 13.17: 自然归并排序

图13.17描述了这一思路。算法开始时，我们从两侧扫描序列，分别找到最长的非递减子序列。然后这两个子序列被归并到一个工作区。归并的结果从工作区的头部依次放置。接着，我们重复这一步骤，继续从两侧向中心进行扫描。这一次，我们将两个已序子序列的结果归并到工作区的右侧，从右向左依次放置。这样的布局可以方便下一轮的扫描。当所有的元素都被扫描并归并到工作区后，我们转而对工作区内的元素进行扫描，而使用原数组作为工作区。每轮都进行这样的切换。最后如有必要，我们将所有的元素从工作区复制到原数组。

唯一的问题是何时结束这一算法。当开始新一轮的扫描时，如果发现最长的非递减子列表一直伸展到数组的末尾，也就是说整个序列已序，此时排序过程结束。

由于这样的归并方式，从头尾两路处理待排序数组，并且使用了子序列的自然元素顺序，它被称为两路自然归并排序。实现这一算法时需要仔细处理。图13.18描述了自然归并排序时的不变性质（invariant）。任何时候，标记 a 之前的元素和标记 d 之后的元素都已被扫描和归并了。我们要将非递减子序列 $[a, b)$ 向右扩展到最长，同时，要将子序列 $[c, d)$ 向左扩展到最长。工作区的不变性质如图中的第二行所示。f 之前的元素和 r 之后的元素都已经处理过（它们可能包含若干已序的子序列）。奇数轮时（第 1、3、5……轮），我们将子序列 $[a, b)$ 和 $[c, d)$ 从 f 起向右归并；偶数轮时（第 2、4、6……轮），我们将子序列从 r 起向左归并。

在命令式环境中，序列用数组保存。在排序开始前，我们申请和数组同样大小的空间作为工作区。指针 a 和 b 一开始指向最左侧，指针 c 和 d 指向最右侧。指针 f 指向工作区的开头，r 指向工作区的结尾。

```
1: function SORT(A)
2:     if |A| > 1 then
3:         n ← |A|
4:         B ← CREATE-ARRAY(n)                    ▷ 创建工作区
5:         loop
6:             [a, b) ← [1, 1)
7:             [c, d) ← [n + 1, n + 1)
```

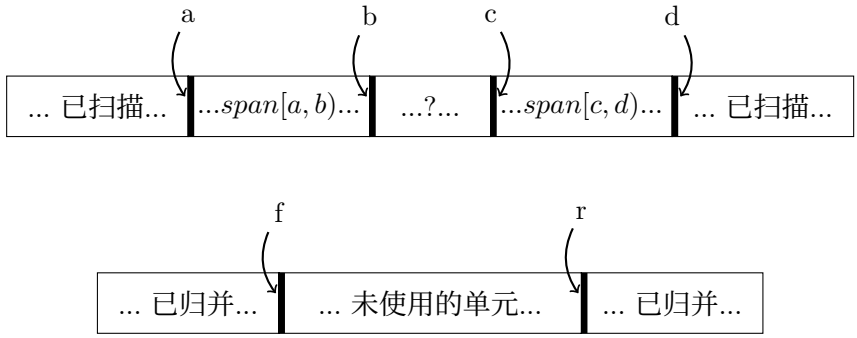


图 13.18: 自然归并排序时的不变性质

```
8:        f ← 1, r ← n                          ▷ 指向工作区首尾的 front 和 rear 指针
9:        t ← False                             ▷ 从 front 归并还是从 rear 归并
10:       while b < c do                        ▷ 存在需要扫描的元素
11:           repeat                            ▷ 扩展 [a, b)
12:               b ← b + 1
13:           until b ≥ c ∨ A[b] < A[b − 1]
14:           repeat                            ▷ 扩展 [c, d)
15:               c ← c − 1
16:           until c ≤ b ∨ A[c − 1] < A[c]
17:           if c < b then                     ▷ 避免 overlap
18:               c ← b
19:           if b − a ≥ n then                 ▷ 若 [a, b) 扩展到整个数组则结束
20:               return A
21:           if t then                         ▷ 从 front 归并
22:               f ← MERGE(A, [a, b), [c, d), B, f, 1)
23:           else                              ▷ 从 rear 归并
24:               r ← MERGE(A, [a, b), [c, d), B, r, −1)
25:           a ← b, d ← c
26:           t ← ¬t                            ▷ 切换归并的方向
27:       EXCHANGE A ↔ B                        ▷ 切换工作区
28:   return A
```

归并算法和此前给出的类似，主要区别在于我们需要将归并的方向作为参数传入。

```
1: function MERGE(A, [a, b), [c, d), B, w, Δ)
2:     while a < b ∧ c < d do
3:         if A[a] < A[d − 1] then
```

```
        B[w] ← A[a]
        a ← a + 1
    else
        B[w] ← A[d − 1]
        d ← d − 1
    w ← w + Δ
while a < b do
    B[w] ← A[a]
    a ← a + 1
    w ← w + Δ
while c < d do
    B[w] ← A[d − 1]
    d ← d − 1
    w ← w + Δ
return w
```

下面的 C 语言例子程序实现了两路自然归并排序算法。这里我们没有释放工作区所申请的内存。

```
int merge(Key* xs, int a, int b, int c, int d, Key* ys, int k, int delta) {
    for(; a < b && c < d; k += delta )
        ys[k] = xs[a] < xs[d-1] ? xs[a++] : xs[--d];
    for(; a < b; k += delta)
        ys[k] = xs[a++];
    for(; c < d; k += delta)
        ys[k] = xs[--d];
    return k;
}

Key* sort(Key* xs, Key* ys, int n) {
    int a, b, c, d, f, r, t;
    if(n < 2)
        return xs;
    for(;;) {
        a = b = 0;
        c = d = n;
        f = 0;
        r = n-1;
        t = 1;
        while(b < c) {
            do {      //扩展[a, b)
                ++b;
            } while( b < c && xs[b-1] ≤ xs[b] );
            do{       //扩展[c, d)
                --c;
            } while( b < c && xs[c] ≤ xs[c-1] );
            if( c < b )
                c = b;   //消除可能的重叠
```

```
            if( b - a ≥ n)
                return xs;          //已序
            if( t )
                f = merge(xs, a, b, c, d, ys, f, 1);
            else
                r = merge(xs, a, b, c, d, ys, r, -1);
            a = b;
            d = c;
            t = !t;
        }
        swap(&xs, &ys);
    }
    return xs;
}
```

自然归并排序的性能和子数组中元素间的顺序相关。但在实际中，即使在最坏情况下，自然归并排序的性能仍然很好。假设我们运气很差，在第一轮扫描数组时，非递减子序列的长度总为 1。这轮扫描结束后，工作区中归并的已序子数组的长度为 2。假设接下来一轮运气仍然很差，但是此前的结果保证了非递减子序列的长度不可能小于 2。这一轮过后，工作区将包含长度为 4 的归并结果……重复这一过程，每一轮后，归并的已序子数组的长度都加倍，因此最多进行 $O(\lg n)$ 轮扫描和归并。在每一轮中，所有的元素都被扫描。这一最坏情况下的性能仍然为 $O(n \lg n)$。我们稍后在介绍自底向上的归并排序时，会再次解释这一有趣的现象。

在纯函数环境中，由于底层的数据结构是单向链表，我们无法从首尾两端扫描列表。因此需要用别的方法来实现自然归并排序。

由于待排序列表总是由若干非递减子列表构成，我们可以每次取两个子列表，归并出一个更长的列表。我们重复取出列表，然后归并。这样非递减子列表的数目不断减半，最后将得到唯一的列表，也就是最终排序的结果。这一过程可以形式化为下面的等式。

$$sort(L) = sort'(group(L)) \tag{13.36}$$

其中函数 $group(L)$ 将列表中的元素分组成非递减子列表。它可以被描述如下，前面两条为边界条件。

- 若列表为空，则结果为一个列表，它包含一个空列表作为唯一的元素；
- 若列表中只含有一个元素，结果为一个列表，它包含一个只含有一个元素的列表；
- 否则，比较列表中的前两个元素，如果第一个小于等于第二个，就将第一个元素插入到对剩余元素进行递归分组的第一个子列表中的最前面；否则，创建一个只含有第一个元素的列表，接着对剩余的元素进行递归分组。

$$group(L) = \begin{cases} \{L\} & : & |L| \leq 1 \\ \{\{l_1\} \cup L_1, L_2, ...\} & : & l_1 \leq l_2, \{L_1, L_2, ...\} = group(L') \\ \{\{l_1\}, L_1, L_2, ...\} & : & otherwise \end{cases} \tag{13.37}$$

也可以将分组条件抽象成一个参数，传入一个通用的分组函数中。如下面的 Haskell 例子代码所示[6]。

```
groupBy' :: (a→a→Bool) →[a] →[[a]]
groupBy' _ [] = [[]]
groupBy' _ [x] = [[x]]
groupBy' f (x:xs@(x':_)) | f x x' = (x:ys):yss
                         | otherwise = [x]:r
  where
    r@(ys:yss) = groupBy' f xs
```

和 $sort$ 函数相比，$sort'$ 的参数不是一个待排序的元素列表，而是分组后的一系列子列表。

$$sort'(\mathbb{L}) = \begin{cases} \phi & : & \mathbb{L} = \phi \\ L_1 & : & \mathbb{L} = \{L_1\} \\ sort'(mergePairs(\mathbb{L})) & : & otherwise \end{cases} \tag{13.38}$$

前两条是简单边界情况。如果待排序的子列表为空，则结果显然为空；如果仅含有一个子列表，则排序结束。这一子列表就是最终的排序结果；否则，我们调用函数 $mergePairs$ 每两个子列表一组进行归并，然后递归地调用 $sort'$ 函数。

接下来要定义 $mergePairs$ 函数。顾名思义，它不断将成对的非递减子列表归并成更长的列表。

$$mergePairs(L) = \begin{cases} L & : & |L| \leq 1 \\ \{merge(L_1, L_2)\} \cup mergePairs(L'') & : & otherwise \end{cases} \tag{13.39}$$

如果剩余的子列表少于两个，则处理结束；否则，我们首先将前两个子列表 L_1 和 L_2 归并，然后递归地将剩余在 L'' 中的列表对归并。$mergePairs$ 的结果类型是列表的列表，最终 $sort'$ 函数会将它们连接一个列表。

归并函数 $merge$ 和此前的定义一致。下面的 Haskell 例子程序给出了完整的实现：

```
mergesort = sort' ∘ groupBy' (≤)

sort' [] = []
sort' [xs] = xs
sort' xss = sort' (mergePairs xss) where
  mergePairs (xs:ys:xss) = merge xs ys : mergePairs xss
  mergePairs xss = xss
```

[6]虽然 Haskell 的标准库 Data.List 中包含一个 `groupBy` 函数。但是这里不能使用它。这是因为它接受一个相等测试函数作为参数，必须满足自反性、传递性和对称性。但是我们的比较条件为“小于等于”，并不满足对称性。具体可以参考本书附录 A。

另外，我们可以先取出两个子列表，将它们归并为一个临时结果，然后不断取出下一个子列表，将其归并到临时结果中，直到所有剩余的子列表都归并完。这是一个典型的 fold 过程，详细介绍见附录 A。

$$sort(L) = fold(merge, \phi, group(L)) \tag{13.40}$$

下面的 Haskell 例子程序实现了这一用 fold 定义的归并排序：

```
mergesort' = foldl merge [] ∘ groupBy' (≤)
```

练习 13.6

- 使用 fold 实现的自然归并排序在性能上和使用 $mergePairs$ 的算法相同么？如果相同，请给出证明；如果不同，哪个更快？

13.11 自底向上归并排序

从自然归并排序的最差情况分析可以引出一个有趣的内容，归并排序既可以自顶向下进行，也可以自底向上进行。自底向上带来的最大好处是可以很方便地用迭代的方式实现。

为了实现自底向上归并排序，首先将待排序序列变成 n 个子列表，每个子列表只包含一个元素。然后我们将每两个相邻的子序列归并，这样就得到了 $\frac{n}{2}$ 个长度为 2 的已序子序列；如果 n 是奇数，最后会剩余一个长度为 1 的子序列。我们重复将相邻的子序列对归并，最后就会得到排序的结果。Knuth 将这种算法称为“直接两路归并排序”（straight two-way merge sort）[51]。图13.19描述了自底向上的归并排序。

和基本归并排序算法以及奇偶归并排序算法不同，我们无需在每次递归时分割列表。整个列表在一开始时被分为 n 个只有一个元素的子列表，然后接下来不断对它们进行归并。

$$sort(L) = sort'(wraps(L)) \tag{13.41}$$

$$wraps(L) = \begin{cases} \phi & : \quad L = \phi \\ \{\{l_1\}\} \cup wraps(L') & : \quad otherwise \end{cases} \tag{13.42}$$

当然 $wraps$ 也可以使用 map 来实现，具体参见附录 A。

$$sort(L) = sort'(map(\lambda_x \cdot \{x\}, L)) \tag{13.43}$$

我们可以复用自然归并排序中定义的 $sort'$ 函数和 $mergePairs$ 函数。不断成对归并子列表，直到最后只剩下一个列表。

下面的 Haskell 例子程序实现了这一算法。

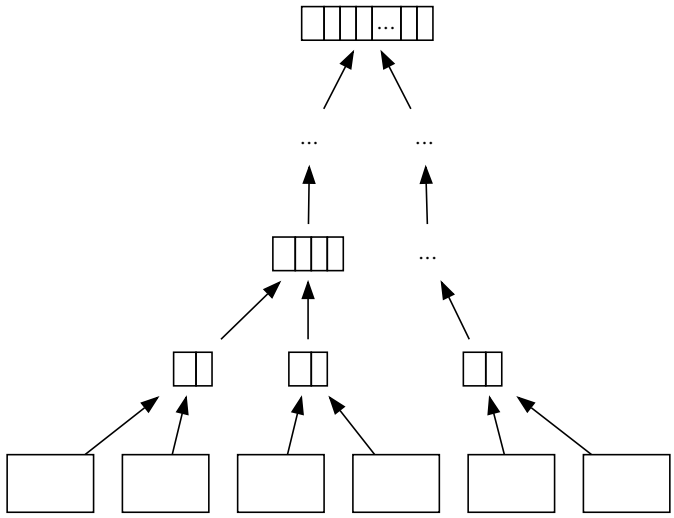


图 13.19: 自底向上归并排序

```
sort = sort' ∘ map (λx→[x])
```

这一算法基于 Okasaki 在 [3] 中给出结果。他和自然归并排序非常类似，仅仅是分组的方法不同。本质上，它可以由自然归并排序的一种特殊情况（最差情况）推导出来：

$$sort(L) = sort'(groupBy(\lambda_{x,y} \cdot False, L)) \tag{13.44}$$

自然归并排序总是将非递减子列表扩展到最长，与此不同，这里的判断条件永远是 False，因此子列表的长度仅扩展到 1 个元素。

和自然归并排序类似，自底向上归并排序也可以用 fold 来定义。具体的实现留给读者作为练习。

观察自底向上归并排序，它已经是尾递归形式了，可以很容易地消除递归，转换成纯迭代算法。

```
1: function SORT(A)
2:     B ← φ
3:     for ∀a ∈ A do
4:         B ← APPEND({a})
5:     N ← |B|
6:     while N > 1 do
7:         for i ← from 1 to ⌊N/2⌋ do
8:             B[i] ← MERGE(B[2i − 1], B[2i])
9:         if ODD(N) then
10:            B[⌈N/2⌉] ← B[N]
```

```
        N ← ⌈N/2⌉
    if B = φ then
        return φ
    return B[1]
```

下面的 Python 例子程序实现了纯迭代式的自底向上归并排序。

```
def mergesort(xs):
    ys = [[x] for x in xs]
    while len(ys) > 1:
        ys.append(merge(ys.pop(0), ys.pop(0)))
    return [] if ys == [] else ys.pop()

def merge(xs, ys):
    zs = []
    while xs ≠ [] and ys ≠[]:
        zs.append(xs.pop(0) if xs[0] < ys[0] else ys.pop(0))
    return zs + (xs if xs ≠[] else ys)
```

和上面的伪代码相比，它每次从头部取出一对子列表，归并好后追加到尾部。这样就极大地简化了奇数个子列表的处理。

练习 13.7

- 使用 fold 实现函数式的自底向上归并排序。
- 只使用数组下标，实现迭代式的自底向上归并排序。不要使用标准库中提供的工具，如 list 或 vector 等。

13.12 并行处理

在基本快速排序的算法中，当划分完成时，可以并行对两个子序列进行排序。这一策略对归并排序也适用。实际上，并行的快速排序和归并排序算法，并不只使用两个并行的任务对划分好的子序列排序，而是将序列分割成 p 个子序列，其中 p 为处理器的个数。理想情况下，如果我们可以并行在 T' 时间内完成排序，并且满足 $O(n \lg n) = pT'$，就称为“线性加速”(linear speed up)，这样的算法叫做最优化并行算法。

但是，简单地扩展基本快速排序算法，选取 $p-1$ 个 pivot，划分为 p 个子序列，然后并行对它们排序，并不是最优化的。瓶颈出现在划分阶段，我们只能得到平均 $O(n)$ 的性能。

另一方面，简单地将基本归并排序算法扩展为并行时，瓶颈出现在归并阶段。为了达到最优化的并行加速，需要对并行的归并排序和快速排序进行更好的设计。实

际上，归并排序和快速排序的分而治之特性使得它们相对容易进行并行化。Richard Cole 在 1986 年发现了使用 n 个处理器，性能为 $O(\lg n)$ 的并行归并排序算法 [76]。

并行处理是一个巨大而复杂的题目，超出了本书描述“基本算法”的范围。读者可以参考 [76] 和 [77] 了解更详细的内容。

13.13 小结

本章介绍了两种常用的分而治之排序算法：快速排序和归并排序。它们都达到了基于比较的排序算法的性能上限 $O(n \lg n)$。Sedgewick 评价快速排序是 20 世纪发现的最伟大的算法。大量的编程环境都使用快速排序作为内置的排序工具。随着时间推移，某些环境中，特别是那些需要处理动态抽象序列的情况下，序列的模型往往不是简单的数组，它们逐渐转而使用归并排序作为通用的排序工具[7]。

这一现象的原因，可以部分地在本章中找到解释。快速排序在大多数情况下表现优异。它主要依靠交换操作，和其他算法相比，快速排序需要较少的交换操作。但是在纯函数环境中，交换并不是最有效的操作，这是因为底层的数据结构通常是单向链表，而不是向量化的数组。另一方面，归并排序则很适合这类环境，它不需要额外的空间，并且即使在快速排序遇到的最坏情况下，也能保证性能。反之快速排序的性能这时就会退化到平方级别。但是在命令式环境中，归并排序不如快速排序在处理数组时的性能表现。它要么需要额外的空间进行归并，要么需要更多的交换操作作为代价。但在某些情况下无法保证有足够的空间可用，例如在嵌入式系统中，内存往往受到限制。目前，原地归并排序仍然是一个活跃的研究领域。

虽然本章的题目叫做“快速排序和归并排序”，但这并不是说这两种排序彼此无关。快速排序可以被看作树排序的一种优化形式。同样归并排序也可以由树排序推导出来 [75]。

存在多种对排序算法的分类，常见的如 [51]，另外一种是根据划分的难易程度和归并的难易程度分类 [72]。

例如快速排序，它的归并很容易，因为 pivot 前的子序列中的所有元素，都小于等于 pivot 后子序列中的任意元素。快速排序的归并过程实际上就是序列的简单连接。

与此相反，归并排序的归并过程要比快速排序复杂得多。但是划分过程却很简单。无论是等分成两个子序列、奇偶分割、自然分割、还是自底向上分割。和归并排序相比，快速排序很难保证完美分割。我们在理论上证明了，快速排序无法完全避免最差情况，尽管人们想出一些工程实践方法如 median-of-three，随机快速排序，以及三路划分等。

到本章为止，我们给出了一些基本的排序算法，包括插入排序、树排序、选择排序、堆排序、快速排序和归并排序。排序仍然是计算机科学中活跃的研究领域。在写这一章的时候，人们正经历着当时所谓“大数据”（big data）的挑战，传统的排序方

[7]实际中，大部分排序工具都是某种混合算法，在序列较短时使用插入排序来保持良好的性能

法无法在有限的时间和资源下处理越来越巨大的数据。在某些领域，处理几百 G 的数据已经成为了日常工作中的任务。

练习 13.8

- 使用归并排序的策略，设计一种算法可以从一个序列产生一棵二叉搜索树。

第十四章　搜索

14.1　简介

搜索是一个巨大并且重要的领域。计算机使很多困难的搜索问题得以实现。某些问题由人来解决几乎是不可能的。现代工业机器人可以在生产线旁的一堆零件中找出正确的进行组装；带有全球卫星导航系统（GPS）的汽车可以在地图中找到前往目的地的最佳路线。带有地图和导航系统的现代手机还能搜索到最便宜的购物方案。

本章介绍基本搜索算法中最简单的内容。计算机的一大优点就是可以在巨大的序列中进行暴力扫描。我们通过两个题目来介绍分而治之的搜索策略：一个是在未排序的序列中寻找第 k 大的元素；另一个是在已序序列中进行二分查找。我们还将介绍多维数据中的二分查找。

文本搜索是日常生活中的重要应用。本章介绍两种常见的文本搜索算法：Knuth-Morris-Pratt（简称 KMP）算法，和 Boyer-Moore 算法。它们体现了另一种重要的搜索策略——信息重用。

除了序列搜索，我们还会介绍一些基本算法用来寻找某些问题的解。它们被广泛用于早期的人工智能领域，包括深度优先搜索（DFS）和广度优先搜索（BFS）。

最后我们会简单介绍动态规划，用于寻找问题的最优解。我们同时会介绍贪心算法，它特别适合用来解决某些特定问题。

14.2　序列搜索

虽然现代计算机可以高速地进行暴力查找，即使假设“摩尔定律”被严格遵守，数据增长的速度还是远远超过暴力查找的能力。在本书的最开始，我们就介绍了这样的例子。这就是人们为何不断研究计算机搜索算法的原因。

14.2.1　分而治之的搜索

分而治之是一种常用的解法。我们可以不断地缩小搜索范围，丢弃无需查找的数据。这样就能显著提高搜索的速度。

k 选择问题

考虑在 n 个元素中寻找第 k 小的元素。最直观的想法是先找到最小的一个，将其丢弃，然后在剩余元素中寻找第二小的元素。重复这一寻找最小值再丢弃的步骤 k 次就可以找到第 k 小的元素。在 n 个元素中寻找最小的元素是线性时间 $O(n)$ 的。因此这一方法的性能为 $O(kn)$。

另一种方法是使用我们此前介绍过的堆（heap）数据结构。无论何种堆，例如使用数组实现的隐式二叉堆、斐波那契堆或其它堆，获取堆顶元素再弹出的性能通常为 $O(\lg n)$。因此这一方法，如式 (14.1) 和 (14.2) 所示，找到第 k 小元素的性能为 $O(k \lg n)$。

$$top(k, L) = find(k, heapify(L)) \tag{14.1}$$

$$find(k, H) = \begin{cases} top(H) & : \quad k = 0 \\ find(k-1, pop(H)) & : \quad otherwise \end{cases} \tag{14.2}$$

但是，使用堆的解法相对比较复杂。是否存在有一种简单、快速的方法能找到第 k 小的元素呢?

我们可以使用分而治之的方法来解决这一问题。如果将全部元素划分为两个子序列 A 和 B，使得 A 中的全部元素都小于等于 B 中的任何元素，我们就可按照下面的方法减小问题的规模[1]:

1. 比较子序列 A 的长度和 k 的大小;
2. 若 $k < |A|$，则第 k 小的元素必然在 A 中，我们可以丢弃子序列 B，然后在 A 中进一步查找;
3. 若 $|A| < k$，则第 k 小的元素必然在 B 中，我们可以丢弃子序列 A，然后在 B 中进一步查找 第 $(k - |A|)$ 小的元素。

注意下划线 部分强调了递归的特性。理想情况下，我们总是将序列划分为相等长度的两个子序列 A 和 B，这样每次都将问题的规模减半，因此性能为线性时间 $O(n)$。

关键问题是如何实现划分，将前 m 小的元素放入一个子序列中，将剩余元素放入另一个中。

回忆快速排序中的划分算法，它将所有小于 pivot 的元素移动到前面，将大于 pivot 的元素移动到后面。根据这一思路，我们可以构造一个分而治之的 k 选择算法，称为“快速选择算法”。

1. 随机选择一个元素（例如第一个）作为 pivot;
2. 将所有不大于 pivot 的元素放入子序列 A；将剩余元素放入子序列 B;

[1]这需要给出一个序列 L 中第 k 小的元素的精确定义：它等于序列 L' 中的第 k 个元素，其中 L' 是 L 的一个排列，并且 L' 满足单调非递减的顺序。

3. 比较 A 的长度和 k，若 $|A| = k - 1$，则 pivot 就是第 k 小的元素；

4. 若 $|A| > k - 1$，递归在 A 中寻找第 k 小的元素；

5. 否则，递归在 B 中寻找第 $(k - 1 - |A|)$ 小的元素；

这一算法可以形式化为下面的等式。设 $0 < k \leq |L|$，其中 L 是一个非空列表。记 l_1 为 L 中的第一个元素，它被选作 pivot；L' 包含除 l_1 外的剩余元素。$(A, B) = partition(\lambda_x \cdot x \leq l_1, L')$。函数 $partition$ 使用快速排序中介绍的算法将 L' 划分为两部分。

$$top(k, L) = \begin{cases} l_1 & : & |A| = k - 1 \\ top(k - 1 - |A|, B) & : & |A| < k - 1 \\ top(k, A) & : & otherwise \end{cases} \tag{14.3}$$

$$partition(p, L) = \begin{cases} (\phi, \phi) & : & L = \phi \\ (\{l_1\} \cup A, B) & : & p(l_1), (A, B) = partition(p, L') \\ (A, \{l_1\} \cup B) & : & \neg p(l_1) \end{cases} \tag{14.4}$$

下面的 Haskell 例子程序实现了这一算法。

```
top n (x:xs) | len == n - 1 = x
             | len < n - 1 = top (n - len - 1) bs
             | otherwise = top n as
    where
      (as, bs) = partition (≤x) xs
      len = length as
```

Haskell 的标准库中提供了 `partition` 函数，具体实现可以参考前面关于快速排序的章节。

最幸运的情况下，第 k 个元素一开始就恰好被选为 pivot。划分函数检查全部列表，发现有 $k - 1$ 个元素不大于 pivot，搜索在 $O(n)$ 时间完成。最差情况下，每次都选择了待查找序列中的最大值或者最小值作为 pivot。划分的结果中，A 或者 B 之一总有一个为空。如果每次总选择最小的元素作为 pivot，则性能为 $O(kn)$。如果每次总选择最大的元素作为 pivot，则性能为 $O((n - k)n)$。

最好情况（不是最幸运情况）是每次 pivot 恰好完美划分列表。A 的长度和 B 的长度几乎相同。序列每次减半。这样总共需要 $O(\lg n)$ 次划分，每次划分的时间和不断减半的序列长度成正比。因此总体性能为 $O(n + \frac{n}{2} + \frac{n}{4} + ... + \frac{n}{2^m})$，其中 m 是满足不等式 $\frac{n}{2^m} < k$ 的最小整数。对上述序列求和结果为 $O(n)$。

平均情况的性能分析需要使用数学期望。方法和快速排序的平均性能分析类似。我们将其作为练习留给读者。和快速排序类似，这一分而治之的选择算法在实际中的绝大部分情况下表现良好。我们可以使用和快速排序中同样的工程方法，例如三点中值法（median-of-three）或随机 pivot 选择来减少最差情况的发生。如下面的命令式实现所示：

```
1: function TOP(k, A, l, u)
2:     EXCHANGE A[l] ↔ A[ RANDOM(l, u) ]          ▷ 随机在范围 [l, u] 内选择
3:     p ← PARTITION(A, l, u)
4:     if p − l + 1 = k then
5:         return A[p]
6:     if k < p − l + 1 then
7:         return TOP(k, A, l, p − 1)
8:     return TOP(k − p + l − 1, A, p + 1, u)
```

这一算法在数组 A 的闭区间 $[l, u]$ 范围内（包括边界上的元素）搜索第 k 小的元素。首先随机选择一个位置，然后把这一位置上的元素作为 pivot 并和第一个元素交换。划分算法在数组内移动元素，并返回最终 pivot 所在的位置。如果 pivot 的最终位置恰好是 k，则搜索结束；如果不大于 pivot 的元素个数多于 $k-1$ 个，算法就递归在范围 $[l, p-1]$ 内搜索第 k 小的元素；否则，我们从 k 中减去不大于 pivot 的元素个数，然后递归在 $[p+1, u]$ 内搜索。

有多种方法可以用来实现划分算法，例如下面给出的基于 N. Lomuto 方法的实现。其它实现我们作为练习留给读者。

```
1: function PARTITION(A, l, u)
2:     p ← A[l]
3:     L ← l
4:     for R ← l + 1 to u do
5:         if ¬(p < A[R]) then
6:             L ← L + 1
7:             EXCHANGE A[L] ↔ A[R]
8:     EXCHANGE A[L] ↔ p
9:     return L
```

下面的 C 语言例子程序实现了这一算法。它处理了某些特殊的情况。一种是数组为空的情况，另一种是 k 超出了数组边界的情况。这些情况下它返回-1 表示搜索失败。

```
int partition(Key* xs, int l, int u) {
    int r, p = l;
    for (r = l + 1; r < u; ++r)
        if (!(xs[p] < xs[r]))
            swap(xs, ++l, r);
    swap(xs, p, l);
    return l;
}

//结果保存在xs[k]中，若u - l ≥ k返回k，否则返回 -1。
int top(int k, Key* xs, int l, int u) {
    int p;
```

```
    if (l < u) {
        swap(xs, l, rand() % (u - l) + l);
        p = partition(xs, l, u);
        if (p - l + 1 == k)
            return p;
        return (k < p - l + 1) ? top(k, xs, l, p) :
                                 top(k- p + l - 1, xs, p + 1, u);
    }
    return -1;
}
```

Blum、Floyd、Pratt、Rivest 和 Tarjan 在 1973 年给出了一个方法，可以保证在最差情况下的性能仍然为 $O(n)$[4]、[81]。它将列表划分为若干小组，每组最多 5 个元素。每组的中值（median）可以很快确定。这样总共选出 $\frac{n}{5}$ 个中值。我们重复这一步骤，再将选出的值分成若干不超过五个元素的组，并选出“中值的中值”（median of median）。显然可以在 $O(\lg n)$ 时间内选出最终“真正”的中值，这是划分列表的最佳 pivot。接下来，我们用这一 pivot 划分列表，将问题规模缩小一半，然后递归寻找第 k 小的元素。性能可以计算如下：

$$T(n) = c_1 lgn + c_2 n + T(\frac{n}{2}) \tag{14.5}$$

其中 c_1 是计算“中值的中值”的常数系数，c_2 是划分的常数系数。可以使用裂项求和（telescoping）方法解此方程，或者直接用主定理（master theorem）[4] 得到性能为 $O(n)$。

如果需要选出前 k 小的元素，而无需关心它们的具体顺序，我们通过可以调整上面的算法来满足这一需要：

$$tops(k, L) = \begin{cases} \phi & : & k = 0 \vee L = \phi \\ A & : & |A| = k \\ A \cup \{l_1\} \cup tops(k - |A| - 1, B) & : & |A| < k \\ tops(k, A) & : & otherwise \end{cases} \tag{14.6}$$

其中 A、B 的定义和此前一样，若 L 不为空，则：$(A, B) = partition(\lambda_x \cdot x \leq l_1, L')$。下面的 Haskell 例子程序实现了这一算法。

```
tops _ [] = []
tops 0 _  = []
tops n (x:xs) | len ==n = as
              | len < n  = as ++ [x] ++ tops (n-len-1) bs
              | otherwise = tops n as
    where
      (as, bs) = partition (≤ x) xs
      len = length as
```

二分查找

二分查找是另一种常见的分而治之算法。我们曾经在插入排序一章提到过。我的中学数学老师曾经表演过这样的“魔术”：我首先想好一个不大于 1000 的数，不说出来。然后他接下来问我一些问题，我只需要回答是或者不是。他需要在十个问题之内猜出那个数。他通常会问这样一些问题：

- 是偶数么？
- 是素数么？
- 所有位上的数字都相同么？
- 能被 3 整除么？
- ……

大多数情况下，我的数学老师总能在十个问题内猜到答案。我和同学们都感到很惊奇。

曾经有一段时间，电视里热播这样的价格竞猜节目。主持人展示一件商品，然后现场的幸运观众需要在 30 秒内猜出价格。对于每次猜测，主持人告知是猜高了，还是猜低了。如果观众能够在 30 秒内猜到正确价格，就可以拿走商品。最好的竞猜策略就是分而治之的二分查找。我们常常可以看到下面这样的猜测和反馈：

- 观众：1000 元；
- 主持人：高了；
- 观众: 500 元；
- 主持人：低了；
- 观众：750 元；
- 主持人：低了；
- 观众：890 元；
- 主持人：低了；
- 观众：990 元；
- 主持人：正确！

我的数学老师说，因为数字不大于 1000，如果通过设计良好的问题，每次能排除一半可能的数字，就可以在 10 次内找出答案。这是因为 $2^{10} = 1024 > 1000$。但是，

如果简单地问“比 500 大么？比 250 小么？……”就太枯燥了。而问题“是偶数么?”就是一个非常好的问题，它总是能去掉一半的数字[2]。

回到二分查找的问题上。它只能在已序的序列中进行查找。我曾经看到有人试图对未排序的数组进行二分查找，花了几个小时也没有搞清楚为什么不正确。二分查找的思路很直观，为了在已序序列 A 中寻找数字 x，我们首先检查中点上的数字，和 x 进行比较，如果恰好相等，则它就是答案，查找结束；如果 x 较小，由于 A 是已序的，我们只需要在前半部分中继续查找；否则，我们在后半部分中继续查找。如果当 A 变成空序列，而我们仍未找到 x，则说明 x 不存在序列中。

在给出形式化的算法定义前，有一个很令人吃惊的事实。高德纳（Donald Knuth）指出：“虽然二分查找的基本思想相对直观，具体细节却复杂得不可思议……”。Jon Bentley 指出，大多数二分查找的实现中含有错误。并且他本人在《编程珠玑》（Programming pearls）第一版中给出实现也隐藏了一个错误，直到 20 多年后才被发现[2]。

二分查找有两种实现，一种是递归的，另一种是迭代的。上面给出的描述，实际就是递归的解法。令数组的上下界分别为 l 和 u，不包含 u 位置上的元素。

```
1: function BINARY-SEARCH(x, A, l, u)
2:     if u < l then
3:         Not found error
4:     else
5:         m ← l + ⌊(u − l)/2⌋                    ▷ 避免计算 ⌊(l + u)/2⌋ 溢出
6:         if A[m] = x then
7:             return m
8:         if x < A[m] then
9:             return BINARY-SEARCH(x, A, l, m - 1)
10:        else
11:            return BINARY-SEARCH(x, A, m + 1, u)
```

如注释中强调的，因为使用有限的字节表示整数，我们不能简单地用 $\lfloor \frac{l+u}{2} \rfloor$ 来计算中点，如果 l 和 u 很大，可能会造成溢出。

二分查找也可以用迭代的方式实现，根据中点上数字比较的结果，我们不断更改待搜索范围的边界。

```
1: function BINARY-SEARCH(x, A, l, u)
2:     while l < u do
3:         m ← l + ⌊(u − l)/2⌋
4:         if A[m] = x then
5:             return m
6:         if x < A[m] then
```

[2]在作者修订本章内容时，微软在社交网络上公布了一个游戏。用户可以想出一个人，然后人工智能机器人向用户提 16 个问题，用户只需要回答是或者不是，最后机器人能说出用户所想的人是谁。你能分析出这个机器人的工作原理么?

```
7:            u ← m − 1
8:        else
9:            l ← m + 1
    return NIL
```

实现二分查找是一个很好的练习。我们把它留给读者，请尝试用各种方法来验证程序的正确性。

由于每次都将待查找数组缩短一半，二分查找的性能为 $O(\lg n)$。

在纯函数式环境中，列表本质上是单向链表。随机访问指定位置的元素需要线性时间。二分查找无法发挥它的优势。下面的分析给出了性能会怎样下降。考虑下面的定义：

$$bsearch(x, L) = \begin{cases} Err & : & L = \phi \\ b_1 & : & x = b_1, (A, B) = splitAt(\lfloor \frac{|L|}{2} \rfloor, L) \\ bsearch(x, A) & : & B = \phi \vee x < b_1 \\ bsearch(x, B') & : & otherwise \end{cases}$$

其中 b_1 是列表 B 不为空时的第一个元素，B' 包含除 b_1 外的剩余部分。函数 $splitAt$ 需要 $O(n)$ 时间将列表分成两个子列表 A 和 B（参见附录 A 和归并排序一章）。若 B 不为空，且 x 等于 b_1，则搜索结束；如果 x 小于 b_1，由于列表已序，我们需要递归在 A 中搜索，否则，需要在 B 中搜索。如果列表为空，则表示搜索失败，待查找的元素不存在。

由于总是在中点位置分割列表，每次递归都将待搜索的元素减半。在每次递归中，都需要线性时间进行分割。分割函数只需要遍历单向链表的前半部分，因此总时间可以表示为：

$$T(n) = c\frac{n}{2} + c\frac{n}{4} + c\frac{n}{8} + ...$$

这一结果为 $O(n)$，和从头至尾进行扫描的结果是一样的。

$$search(x, L) = \begin{cases} Err & : & L = \phi \\ l_1 & : & x = l_1 \\ search(x, L') & : & otherwise \end{cases}$$

在插入排序一章中，我们曾经指出，函数式的二分查找本质上是通过二叉搜索树实现的。将已序序列表示为一棵树（如有必要使用自平衡树），可以提供对数时间的搜索[3]

虽然无法对单向链表进行分而治之的二分查找，但二分查找在函数式环境中也有很多应用。考虑方程 $a^x = y$，对于给定的自然数 a 和 y，其中 $a \leq y$。我们希望寻找 x 的整数解。显然可以用穷举搜索：从 0 开始依次尝试 $a^0, a^1, a^2, ...$，直到发现某

[3]有些读者认为，应该使用数组而不是单向链表，例如 Haskell 中提供了能在常数时间进行随机访问的数组。本书只讨论用手指树实现的纯函数式序列，和 Haskell 中的数组不同，它并不支持常数时间的随机访问。

个 $a^i = y$，或者发现 $a^i < y < a^{i+1}$，这表示方程无整数解。我们定义解 x 的范围为 $X = \{0, 1, 2, ...\}$，并且定义下面的穷举搜索函数 $solve(a, y, X)$。

$$solve(a, y, X) = \begin{cases} x_1 & : & a^{x_1} = y \\ solve(a, y, X') & : & a^{x_1} < y \\ Err & : & otherwise \end{cases}$$

这一函数按照单调增的顺序检查解的可能范围。它首先从 X 选择一个候选元素 x_1，比较 a^{x_1} 和 y，如果相等，则 x_1 就是方程的解；如果小于 y，则丢弃 x_1，继续在剩余的元素 X' 中查找；否则，由于函数 $f(x) = a^x$ 在 a 为自然数时，是非减函数，剩余元素会令 $f(x)$ 变得更大，因此方程不存在整数解。这种情况下我们返回错误。

对于很大的 a 和 x，如果需要保持精度，则计算 a^x 会消耗一定的时间[4]。有没有什么办法可以减小计算量呢？我们可以使用分而治之的二分查找来进行改进。我们可以估计出解的范围的上限。由于 $a^y \geq y$，我们可以在区间 $\{0, 1, ..., y\}$ 内搜索。由于函数 $f(x) = a^x$ 是非减函数，对于自变量 x，我们可以先检查区间的中点 $x_m = \lfloor \frac{0+y}{2} \rfloor$，如果 $a^{x_m} = y$，则 x_m 就是方程的解；如果值小于 y，我们可以丢弃 x_m 前的全部元素；否则，我们丢弃 x_m 后的全部元素；两种情况下都将搜索范围减半。我们重复这一过程直到找到解或者查找范围变成空，这表示方程不存在整数解。

二分查找的方法可以形式化为下面式 (14.7) 的定义。其中，我们将非减函数抽象为一个参数。为了解决上面的方程，我们只需要调用 $bsearch(f, y, 0, y)$，其中 $f(x) = a^x$。

$$bsearch(f, y, l, u) = \begin{cases} Err & : & u < l \\ m & : & f(m) = y, m = \lfloor \frac{l+u}{2} \rfloor \\ bsearch(f, y, l, m-1) & : & f(m) > y \\ bsearch(f, y, m+1, u) & : & f(m) < y \end{cases} \tag{14.7}$$

由于我们每次递归都将搜索范围减半，这一方法只计算了 $O(\log y)$ 次 $f(x)$。要远好于穷举法。

二维搜索

一个很自然的想法是把二分查找的思想扩展到二维括者更高维的搜索域。但事实上这种扩展却并不简单。

作为一个例子，考虑一个 $m \times n$ 矩阵 M。每行、每列的元素都是严格递增的。图14.1给出了一个这样的矩阵。

任给一个 x，如何快速地在矩阵中定位到所有等于 x 的元素呢？我们需要给出一个算法，返回一组位置 (i, j) 的列表，使得所有的 $M_{i,j} = x$。

Richard Bird 说他曾经用这一问题作为牛津大学的入学面试题 [1]。耐人寻味的是，那些在中学就接触过计算机科学的候选人，往往会尝试使用二分查找来解决这个问题，但却很容易陷入困境。

[4]当然，我们可以复用 a^n 的结果来计算 $a^{n+1} = aa^n$。这里我们考虑一般意义下的单调函数 $f(n)$。

$$\begin{bmatrix} 1 & 2 & 3 & 4 & \dots \\ 2 & 4 & 5 & 6 & \dots \\ 3 & 5 & 7 & 8 & \dots \\ 4 & 6 & 8 & 9 & \dots \\ \dots & & & & \end{bmatrix}$$

图 14.1: 每行、每列都严格单调增的矩阵

按照二分查找的思路，通常会先检查位于 $M_{\frac{m}{2},\frac{n}{2}}$ 上的元素。如果它小于 x，我们只能丢弃左上区域的元素；如果它大于 x，只能丢弃右下区域的元素。图14.2描述了这两种情况，灰色的区域表示可以丢弃的元素。

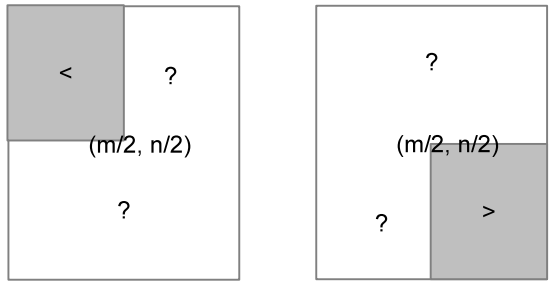


图 14.2: 左：中点的元素小于 x。所有灰色区域的元素都小于 x；右：中点的元素大于 x。所有灰色区域的元素都大于 x

这里出现的问题是，两种情况下，搜索区域都从一个矩形变成了一个 L 形，我们无法继续递归地进行搜索。为了系统化地解决这一问题，我们先给出一个通用的定义，然后从穷举法开始，逐步改进，直到获得满意的答案。

考虑严格单调增函数 $f(x, y)$，例如 $f(x, y) = a^x + b^y$，其中 a 和 b 都是自然数。给定自然数 z，我们希望寻找全部的非负整数解 (x, y)。

使用这一定义，上述的矩阵搜索问题，可以特殊化为下面的函数：

$$f(x, y) = \begin{cases} M_{x,y} & : & 1 \leq x \leq m, 1 \leq y \leq n \\ -1 & : & otherwise \end{cases}$$

穷举法二维搜索

既然要找出 $f(x, y)$ 的所有解，最简单的方法就是双重循环的穷举法：

```
1: function SOLVE(f, z)
2:     A ← φ
```

```
3:    for x ∈ {0, 1, 2, ..., z} do
4:        for y ∈ {0, 1, 2, ..., z} do
5:            if f(x, y) = z then
6:                A ← A ∪ {(x, y)}
7:    return A
```

显然，这一方法计算了 $(z+1)^2$ 次 f。它可以形式化为式 (14.8) 的定义：

$$solve(f, z) = \{(x, y) | x \in \{0, 1, ..., z\}, y \in \{0, 1, ..., z\}, f(x, y) = z\} \tag{14.8}$$

Saddleback 搜索

我们尚未使用 $f(x, y)$ 为严格单调增的条件。Dijkstra 指出 [82]，有效的解法不是从左下角出发，而是从左上角出发开始查找。如图14.3所示，搜索从 $(0, z)$ 开始，对于每个点 (p, q)，我们比较 $f(p, q)$ 和 z 的关系：

- 如果 $f(p, q) < z$，由于 f 单调增，对于所有的 $0 \leq y < q$，必然有 $f(p, y) < z$。我们可以丢弃垂直线段上的所有点（红色线段）；
- 如果 $f(p, q) > z$，则对于所有的 $p < x \leq z$，必然有 $f(x, q) > z$。我们可以丢弃水平线段上的所有点（蓝色线段）；
- 否则，若 $f(p, q) = z$，则 (p, q) 是一个解，两条线段上的点都可以丢弃。

这样，我们就可以逐步缩小矩形的搜索区域。每次要么丢弃一行，要么丢弃一列，或者同时丢弃行和列。

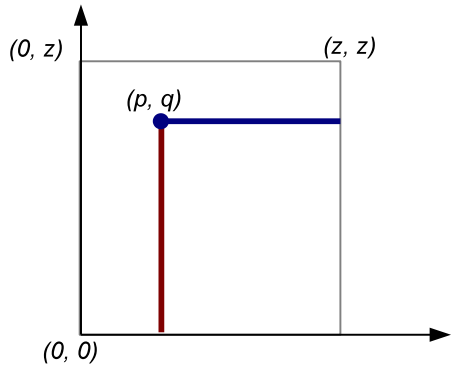


图 14.3: 从左上角搜索

这一方法可以定义为一个函数 $search(f, z, p, q)$,它在矩形区域内搜索方程 $f(x, y) = z$ 的整数解，矩形的左上角为 (p, q)，右下角为 $(z, 0)$。这个矩形的左上角一开始时为

$(p, q) = (0, z)$，然后启动搜索 $solve(f, z) = search(f, z, 0, z)$。

$$search(f, z, p, q) = \begin{cases} \phi & : & p > z \vee q < 0 \\ search(f, z, p + 1, q) & : & f(p, q) < z \\ search(f, z, p, q - 1) & : & f(p, q) > z \\ \{(p, q)\} \cup search(f, z, p + 1, q - 1) & : & otherwise \end{cases} \tag{14.9}$$

第一行为边界条件，如果 (p, q) 不在 $(z, 0)$ 的左上方，则无解。下面的 Haskell 例子程序实现了这一算法：

```
solve f z = search 0 z where
  search p q | p > z || q < 0 = []
             | z' < z = search (p + 1) q
             | z' > z = search p (q - 1)
             | otherwise = (p, q) : search (p + 1) (q - 1)
    where z' = f p q
```

考虑到计算 f 的过程消耗可能较大，这一程序将计算结果 $f(p, q)$ 存储在变量 z' 中。算法也可以用 imperative 的方式实现，在循环中不断更新搜索区域的边界。

1: **function** SOLVE(f, z)
2: $p \leftarrow 0, q \leftarrow z$
3: $S \leftarrow \phi$
4: **while** $p \leq z \wedge q \geq 0$ **do**
5: $z' \leftarrow f(p, q)$
6: **if** $z' < z$ **then**
7: $p \leftarrow p + 1$
8: **else if** $z' > z$ **then**
9: $q \leftarrow q - 1$
10: **else**
11: $S \leftarrow S \cup \{(p, q)\}$
12: $p \leftarrow p + 1, q \leftarrow q - 1$
13: **return** S

下面的 Python 例子程序实现了这一算法。

```
def solve(f, z):
    (p, q) = (0, z)
    res = []
    while p ≤ z and q ≥ 0:
        z1 = f(p, q)
        if z1 < z:
            p = p + 1
        elif z1 > z:
            q = q - 1
        else:
            res.append((p, q))
```

```
            (p, q) = (p + 1, q - 1)
    return res
```

显然在每次迭代中，p 和 q 中至少有一个会向右下角前进一步。因此最多需要 $2(z+1)$ 次迭代以完成搜索。这是最差情况下的结果。最好的情况又分为三种，第一种是每次迭代 p 和 q 同时前进一步，因此只需要 $z+1$ 步就可以完成搜索；第二种是不断沿着水平方向向右前进，最后 p 超过 z；第三种与此类似，不断沿着垂直方向向下前进，最终 q 变为负。

图14.4描述了最好和最坏的情况。图14.4 (a) 中，对角线上的每个点 $(x, z-x)$ 都满足 $f(x, z-x) = z$，总共需要 $z+1$ 步到达 $(z, 0)$；(b) 中，最上方水平线上的每个点 (x, z) 都使得 $f(x, z) < z$，$z+1$ 步后，搜索结束；(c) 中，左侧垂直线上的每个点 $(0, x)$ 都使得 $f(0, x) > z$，因此 $z+1$ 步后，搜索结束；(d) 描述的是最差情况。如果我们将搜索路径上的所有水平线段投射 x 轴上，所有垂直线段投射到 y 轴上，就可以得到总共的搜索步数为 $2(z+1)$。

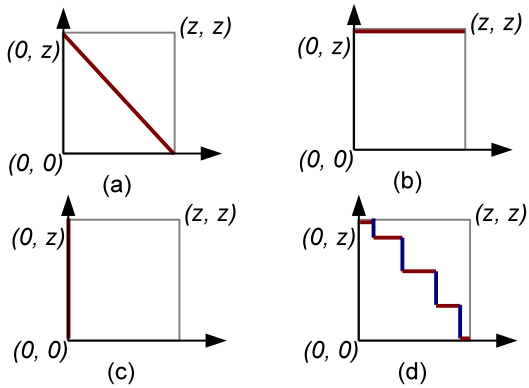


图 14.4: 最好和最差情况

和复杂度为 $O(z^2)$ 的穷举法相比，这一改进将复杂度提高到线性时间 $O(z)$。

Bird 猜测, 这一算法的名称 saddleback 的由来是因为函数 f 的 3 维图像中, 左下部的最小值和右上部的最大值, 以及两侧的翼形图像, 合起来像一个马鞍。如图14.5所示。

改进的 saddleback 搜索

问题扩展到 2 维后，我们尚未使用二分查找来改进算法。基本的 saddleback 搜索从左上角 $(0, z)$ 开始，向右下角 $(z, 0)$ 进行搜索。这一范围实可以进一步缩小。

因为 f 单调增, 我们可以沿着 y 轴找到最大的 m, 使得 $0 \leq m \leq z$ 且 $f(0, m) \leq z$; 同样，我们可以沿着 x 轴找到最大的 n，使得 $0 \leq n \leq z$ 且 $f(n, 0) \leq z$；这样搜索区域就从原来的 $(0, z) - (z, 0)$ 缩小到 $(0, m) - (n, 0)$，如图14.6所示。

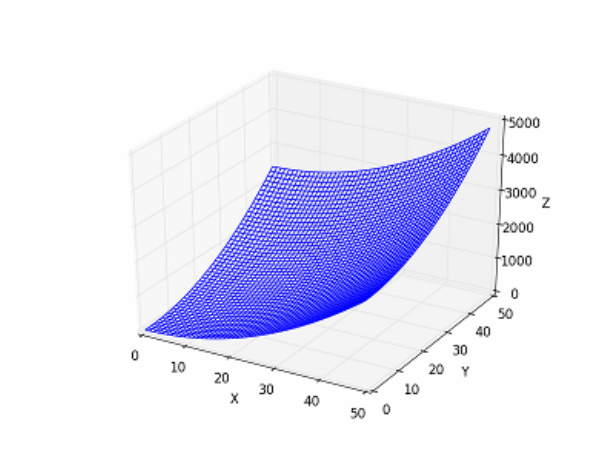


图 14.5: 函数 $f(x, y) = x^2 + y^2$ 的图像

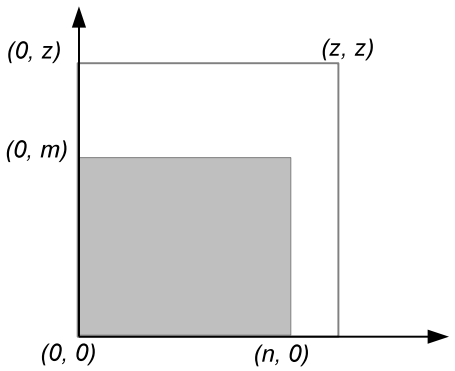


图 14.6: 缩小的灰色搜索区域

显然 m 和 n 可用穷举法找到：

$$\begin{aligned} m &= max(\{y | 0 \le y \le z, f(0, y) \le z\}) \\ n &= max(\{x | 0 \le x \le z, f(x, 0) \le z\}) \end{aligned} \tag{14.10}$$

当搜索 m 时，函数 f 的变量 x 固定为 0。这就转化为了一维的单调增函数搜索问题（在函数式环境中，称为 Curried 化函数 $f(0, y)$）。可以使用二分查找来改进这一搜索。但是我们需要对式 (14.7) 略加改动。给定 y，我们不是要寻找 $l \le x \le u$，使得 $f(x) = y$。而是要寻找 $l \le x \le u$ 使得满足不等式 $f(x) \le y < f(x + 1)$。

$$bsearch(f, y, l, u) = \begin{cases} l & : & u \le l \\ m & : & f(m) \le y < f(m + 1), m = \lfloor \frac{l+u}{2} \rfloor \\ bsearch(f, y, m + 1, u) & : & f(m) \le y \\ bsearch(f, y, l, m - 1) & : & otherwise \end{cases} \tag{14.11}$$

第一行处理搜索区域为空的边界情况，此时我们返回搜索区域的下界；如果中点对应的函数值小于等于 y，而下一个值对应的函数值大于 y，则中点就是我们要搜索的结果；否则，如果中点下一个值对应的函数值也不大于 y，就将中点的下一个值作为新的下届，递归地进行二分查找；最后，如果中点对应的函数值大于 y，则用中点前的一个值作为新的上界，递归进行查找。下面的 Haskell 例子程序实现了这样的二分查找。

```
bsearch f y (l, u) | u ≤ l = l
                   | f m ≤ y = if f (m + 1) ≤ y
                               then bsearch f y (m + 1, u) else m
                   | otherwise = bsearch f y (l, m-1)
  where m = (l + u) `div` 2
```

这样，m 和 n 可以使用二分查找来确定：

$$\begin{aligned} m &= bsearch(\lambda_y \cdot f(0, y), z, 0, z) \\ n &= bsearch(\lambda_x \cdot f(x, 0), z, 0, z) \end{aligned} \tag{14.12}$$

我们可以将 saddleback 搜索的区域缩小为更精确的矩形 $solve(f, z) = search(f, z, 0, m)$：

$$search(f, z, p, q) = \begin{cases} \phi & : & p > n \vee q < 0 \\ search(f, z, p + 1, q) & : & f(p, q) < z \\ search(f, z, p, q - 1) & : & f(p, q) > z \\ \{(p, q)\} \cup search(f, z, p + 1, q - 1) & : & otherwise \end{cases} \tag{14.13}$$

大部分和基本的 saddleback 一样，但是当 p 超过 n 的时候，就可以停止，而无需达到 z。在实际的实现中，可以将 $f(p, q)$ 的值保存下来，而不用每次计算。如下面的 Haskell 例子代码所示：

```
solve' f z = search 0 m where
  search p q | p > n || q < 0 = []
             | z' < z = search (p + 1) q
             | z' > z = search p (q - 1)
             | otherwise = (p, q) : search (p + 1) (q - 1)
    where z' = f p q
  m = bsearch (f 0) z (0, z)
  n = bsearch (λx→f x 0) z (0, z)
```

这一改进的 saddleback 搜索，首先使用两轮二分查找得到 m 和 n。每轮二分查找都计算了 $O(\lg z)$ 次 f；此后，算法在最坏情况下计算 $O(m+n)$ 次；而在最好的情况下计算 $O(min(m,n))$ 次。总体的性能如下表所示：

	计算 f 的次数
最坏情况	$2\log z + m + n$
最好情况	$2\log z + min(m,n)$

表 14.1: 改进 saddleback 搜索的性能

某些函数，例如 $f(x,y) = a^x + b^y$，对于正整数 a 和 b，m 和 n 相对很小，因此整体性能接近 $O(\lg z)$。

这一算法也可以用命令式的方法实现。首先需要修改命令式的二分查找算法：

```
1: function BINARY-SEARCH(f, y, (l, u))
2:     while l < u do
3:         m ← ⌊(l+u)/2⌋
4:         if f(m) ≤ y then
5:             if y < f(m + 1) then
6:                 return m
7:             l ← m + 1
8:         else
9:             u ← m
10:    return l
```

使用上述二分查找，在开始 saddleback 搜索前，先确定 m 和 n。

```
1: function SOLVE(f, z)
2:     m ← BINARY-SEARCH(λ_y · f(0, y), z, (0, z))
3:     n ← BINARY-SEARCH(λ_x · f(x, 0), z, (0, z))
4:     p ← 0, q ← m
5:     S ← φ
6:     while p ≤ n ∧ q ≥ 0 do
7:         z' ← f(p, q)
8:         if z' < z then
```

```
        p ← p + 1
    else if z' > z then
        q ← q − 1
    else
        S ← S ∪ {(p, q)}
        p ← p + 1, q ← q − 1
  return S
```

具体的实现留给读者作为练习。

Saddleback 搜索的进一步改进

图14.2展示的两种情况中，矩阵中点的值要么比目标值小，要么比目标值大。都只能丢弃 $\frac{1}{4}$ 区域中的元素，而剩余的搜索区域变为一个 L 形。

事实上，我们忽略了另外的一个重要情况。我们观察矩形搜索区域中的任一一点，如图14.7所示。

考虑搜索一个矩形区域，左上角为 (a, b)，右下角为 (c, d)。如果 (p, q) 不是矩形的中点，并且 $f(p, q) \neq z$，我们并不能保证被丢弃的部分总是 $1/4$。但是，如果 $f(p, q) = z$，由于 f 是单调增的，我们可以同时丢弃左下和右上的子区域，并且 p 列和 q 行上的所有其他点也都可以丢弃掉。这样每次只剩下 $1/2$ 的区域，可以迅速缩小搜索的区间。

由此可知，我们无需找到矩形的中点进行搜索。更有效的方法是找到函数值等于目标值的点。我们可以沿着矩形中心的水平方向或者垂直方向使用二分查找来定位这样的点。

在线段上进行二分查找的性能和线段的长度成对数关系。我们可以选取水平和垂直方向中较短的中线进行搜索，如图14.8所示。

但是，如果中线上不存在满足 $(p, q) = z$ 的点时如何处理呢？例如，水平中线上不存在这样的点。此时，我们仍然能够找到一点满足 $f(p, q) < z < f(p + 1, q)$。唯一不同之处是我们不能将 p 列和 q 行上的点完全丢弃。

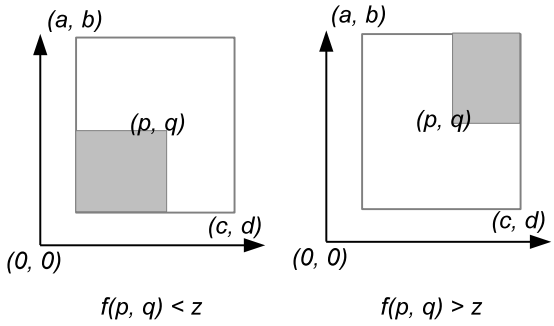
综合上述情况，沿着水平线二分搜索以要找到一点 p，其满足件 $f(p, q) \leq z < f(p + 1, q)$；而沿着垂直线二分搜索的条件是 $f(p, q) \leq z < f(p, q + 1)$。

如果线段上所有的点都使得 $f(p, q) < z$，则修改后的二分查找会返回上界作为结果；反之，如果所有点对应的函数值都大于 z，则返回下界作为结果。此时，我们可以将中线一侧的整个区域全部丢弃。

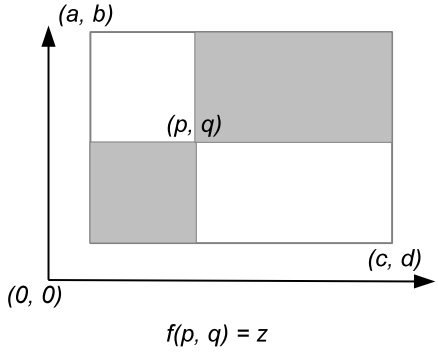
总结这些结论，我们可以给出下面的改进 saddleback 搜索算法：

1. 沿着 y 轴和 x 轴进行二分搜索，定位出搜索区域的边界，从 $(0, m)$ 到 $(n, 0)$；

2. 记待搜索的矩形区域为 $(a, b) - (c, d)$，若矩形为空，则无解；



(a) 如果 $f(p,q) \neq z$，只能丢弃左下或右上的区域（灰色部分）。两种情况下，剩余的搜索区域都变成了 L 形。



(b) 如果 $f(p,q) = z$，可以同时丢弃两个子区域，问题的搜索域减半。

图 14.7: 缩小搜索区域的效率

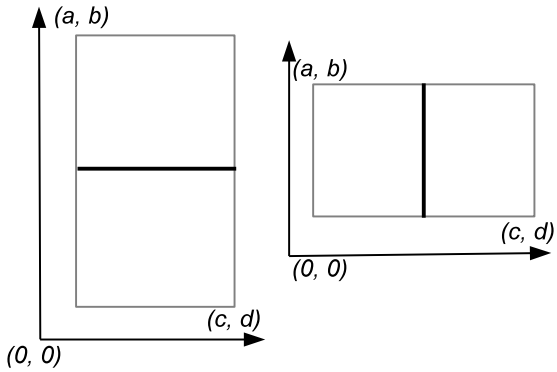


图 14.8: 沿较短的中线进行二分查找

3. 若矩形的高大于宽，则沿着水平中线进行二分查找；否则，沿着垂直中线进行二分查找；记查找的结果为点 (p, q)；

4. 若 $f(p, q) = z$，记录 (p, q) 为一个解，然后递归搜索两个子矩形区域 $(a, b) - (p - 1, q + 1)$ 和 $(p + 1, q - 1) - (c, d)$；

5. 否则，若 $f(p, q) \neq z$，递归搜索同样的两个子矩形区域和一条线段。线段或者为 $(p, q + 1) - (p, b)$ 如图14.9 (a)；或者为 $(p + 1, q) - (c, q)$ 如图14.9 (b)。

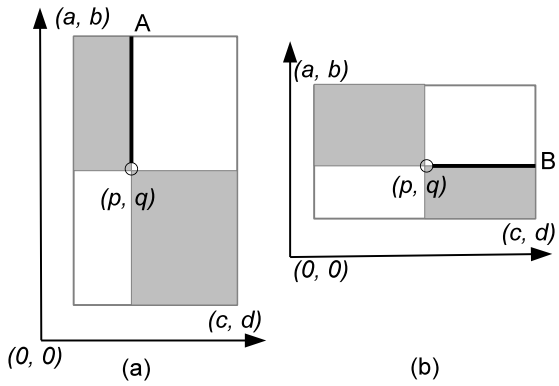


图 14.9: 递归搜索灰色的区域，如果 $f(p, q) \neq z$，还需要搜索加粗的线段

我们复用前面式 (14.11) 和 (14.12) 的定义。定义 $Search_{(a,b),(c,d)}$ 为新的搜索函数，它搜索一个矩形区域，其中左上角为 (a, b)，右下角为 (c, d)。

$$search_{(a,b),(c,d)} = \begin{cases} \phi & : & c < a \vee d < b \\ csearch & : & c - a < b - d \\ rsearch & : & otherwise \end{cases} \tag{14.14}$$

函数 $csearch$ 在水平中线上进行二分查找，寻找一点 (p, q) 使得 $f(p, q) \leq z < f(p + 1, q)$。如图14.9 (a) 所示。如果中线上所有点对应的函数值都大于 z，二分查找返回下界作为结果，即 $(p, q) = (a, \lfloor \frac{b+d}{2} \rfloor)$。中线和它上侧的区域全部可以丢弃，如图14.10 (a). 所示。

$$csearch = \begin{cases} search_{(p,q-1),(c,d)} & : & z < f(p, q) \\ search_{(a,b),(p-1,q+1)} \cup \{(p, q)\} \cup search_{(p+1,q-1),(c,d)} & : & f(p, q) = z \\ search_{(a,b),(p,q+1)} \cup search_{(p+1,q-1),(c,d)} & : & otherwise \end{cases} \tag{14.15}$$

其中

$$\begin{aligned} q &= \lfloor \tfrac{b+d}{2} \rfloor) \\ p &= bsearch(\lambda_x \cdot f(x, q), z, (a, c)) \end{aligned}$$

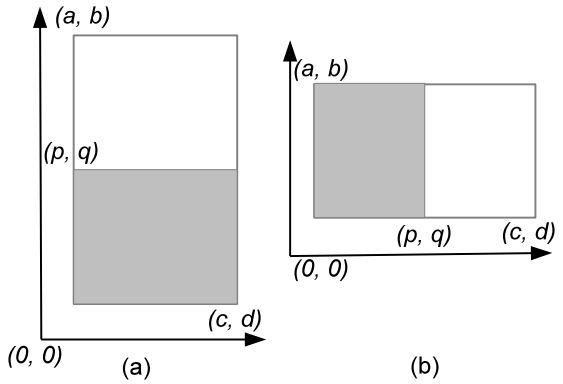


图 14.10: 沿中线进行二分查找时的特殊情况

函数 $rsearch$ 与此类似，它沿着垂直中线进行搜索。

$$rsearch = \begin{cases} search_{(a,b),(p-1,q)} & : \quad z < f(p,q) \\ search_{(a,b),(p-1,q+1)} \cup \{(p,q)\} \cup search_{(p+1,q-1),(c,d)} & : \quad f(p,q) = z \\ search_{(a,b),(p-1,q+1)} \cup search_{(p+1,q),(c,d)} & : \quad otherwise \end{cases} \tag{14.16}$$

其中

$$\begin{aligned} p &= \lfloor \frac{a+c}{2} \rfloor) \\ q &= bsearch(\lambda_y \cdot f(p,y), z, (d,b)) \end{aligned}$$

下面的 Haskell 例子程序实现了这一算法。

```
search f z (a, b) (c, d)
       | c < a || b < d = []
       | c - a < b - d = let q = (b + d) `div` 2 in
           csearch (bsearch (λx → f x q) z (a, c), q)
       | otherwise = let p = (a + c) `div` 2 in
           rsearch (p, bsearch (f p) z (d, b))
  where
    csearch (p, q)
        | z < f p q = search f z (p, q - 1) (c, d)
        | f p q == z = search f z (a, b) (p - 1, q + 1) ++
                (p, q) : search f z (p + 1, q - 1) (c, d)
        | otherwise = search f z (a, b) (p, q + 1) ++
                search f z (p + 1, q - 1) (c, d)
    rsearch (p, q)
        | z < f p q = search f z (a, b) (p - 1, q)
        | f p q == z = search f z (a, b) (p - 1, q + 1) ++
                (p, q) : search f z (p + 1, q - 1) (c, d)
        | otherwise = search f z (a, b) (p - 1, q + 1) ++
                search f z (p + 1, q) (c, d)
```

主程序首先沿着 X 轴和 Y 轴进行二分查找，然后调用上述函数。

```
solve f z = search f z (0, m) (n, 0) where
  m = bsearch (f 0) z (0, z)
  n = bsearch (λx → f x 0) z (0, z)
```

由于每次都丢弃一半的区域，算法总共搜索 $O(\log(mn))$ 轮。但是，为了寻找点 (p, q) 使得问题规模减半，我们需要沿着中线进行二分查找。这样需要计算 f 的次数为 $O(\log min(m, n))$。令在大小为 $m \times n$ 的矩形区域搜索的时间为 $T(m, n)$，我们有如下的递归关系：

$$T(m, n) = \log(min(m, n)) + 2T(\frac{m}{2}, \frac{n}{2}) \tag{14.17}$$

不妨设 $m > n$，使用裂项求和方法，若 $m = 2^i$、$n = 2^j$，我们有：

$$\begin{aligned} T(2^i, 2^j) &= j + 2T(2^{i-1}, 2^{j-1}) \\ &= \sum_{k=0}^{i-1} 2^k(j - k) \\ &= O(2^i(j - i)) \\ &= O(m \log(n/m)) \end{aligned} \tag{14.18}$$

Richard Bird 证明了，这是在 $m \times n$ 的矩形区域内搜索一给定值的最优下界 [1]。命令式的实现与此类似，我们在此将其略过。

练习 14.1

- 参考前面章节快速排序的部分，证明分而治之的 k 选择算法，在平均情况下的性能为 $O(n)$。
- 使用两路划分和三点中值法实现命令式的 k 选择算法。
- 实现能有效处理大量重复元素的命令式的 k 选择算法。
- 选择一门编程语言，实现 median-of-median 的 k 选择算法。
- 本节给出的 $tops(k, L)$ 使用了列表的连接操作，如 $A \cup \{l_1\} \cup tops(k - |A| - 1, B)$。这一操作的性能为线性时间，和被连接列表的长度成比例。修改算法，仅用一遍处理就将子列表连接起来。
- 作者想出了另外一种分而治之的 k 选择问题解法。首先找到前 k 个元素中的最大值，和剩余元素中的最小值，分别记为 x 和 y，若 x 小于 y，说明所有的前 k 个元素都小于剩余的元素，它们恰巧是最小的 k 个元素；否则，说明前 k 个元素中的某些元素，需要被交换到后面去。

```
1: procedure TOPS(k, A)
2:     l ← 1
3:     u ← |A|
```

```
    loop
        i ← MAX-AT(A[l..k])
        j ← MIN-AT(A[k + 1..u])
        if A[i] < A[j] then
            break
        EXCHANGE A[l] ↔ A[j]
        EXCHANGE A[k + 1] ↔ A[i]
        l ← PARTITION(A, l, k)
        u ← PARTITION(A, k + 1, u)
```

请说明这一算法正确与否？性能如何？

- 使用迭代的方式和递归的方式分别实现二分查找算法，并使用自动的方式进行测试。可以使用生成的随机测试数据，也可以定义一些不变性质，并和编程环境中内置的二分查找工具对比。

- 任意给定两个已序数组 A 和 B，寻找它们的中值（median）。要求时间复杂度为 $O(\lg(|A| + |B|))$。

- 使用一门命令式语言，在进行 saddleback 搜索前，先通过二分查找定位出更精确的搜索区域。

- 使用一门命令式语言，沿着较短的中线进行二分查找，从而实现改进的二维搜索。

- 有人给出了这样的二维搜索算法：当搜索一个矩形区域时，由于左下角是最小值，右上角是最大值。若待搜索的值小于最小值或者大于最大值，则无解；否则，从中点将矩形区域分割成 4 个小矩形，然后进行递归搜索。

```
1: procedure SEARCH(f, z, a, b, c, d)          ▷ (a, b): 左下角 (c, d): 右上角
2:     if z ≤ f(a, b) ∨ f(c, d) ≥ z then
3:         if z = f(a, b) then
4:             record (a, b) as a solution
5:         if z = f(c, d) then
6:             record (c, d) as a solution
7:         return
8:     p ← ⌊(a+c)/2⌋
9:     q ← ⌊(b+d)/2⌋
10:    SEARCH(f, z, a, q, p, d)
11:    SEARCH(f, z, p, q, c, d)
12:    SEARCH(f, z, a, b, p, q)
13:    SEARCH(f, z, p, b, c, q)
```

试分析这一算法的性能。

14.2.2 信息复用

人会通过搜索来学习。我们不仅记忆搜索失败的教训，还学习总结成功的模式。这是某种意义上的信息复用，不论这些信息是正面的还是负面的。但难点在于决定记忆哪些信息。记忆太少的信息不足以提高搜索的效率，记忆太多的信息又无法满足存储空间的限制。

本节我们首先介绍两个有趣的问题：Boyer-Moore 众数（majority number）问题，和子数组最大和问题。它们都通过复用最少的信息来解决问题。然后，我们介绍两种被广泛使用的字符串匹配算法：KMP（Knuth-Morris-Pratt）算法，和 Boyer-Moore 算法。

Boyer-Moore 众数问题

人们常常通过投票来进行一些决策，例如选举领袖，接受或者拒绝一项建议。在作者写作本章的时候，有三个国家正在通过投票选举总统，他们都使用计算机来统计投票结果。

假设某个小岛上的国家要通过投票选出新的总统。这个国家的宪法规定，只有赢得半数以上选票的人才可以成为总统。从一个投票结果的序列，例如 A, B, A, C, B, B, D, ... 我们能否找到一种高效的方法，得知谁当选了总统，或者没有任何人赢得半数以上的选票？

显然可以通过使用一个 map，然后遍历一遍选票来解决这个问题。如我们在二叉搜索树一章给出例子那样[5]

```
template<typename T>
T majority(const T* xs, int n, T fail) {
    map<T, int> m;
    int i, max = 0;
    T r;
    for (i = 0; i < n; ++i)
        ++m[xs[i]];
    for (typename map<T, int>::iterator it = m.begin(); it ≠ m.end(); ++it)
        if (it→second > max) {
            max = it→second;
            r = it→first;
        }
    return max * 2 > n ? r : fail;
}
```

[5]2004 年，人们发现了一种概率算法，称为 Count-min sketch 算法，使用 sub-linear 空间进行计数 [84]。

这段例子程序首先扫描所有选票，然后通过 map 累计所有候选人的票数。接着，他遍历 map 找到得票最多的候选人。若票数超过半数，则此人获胜，否则程序返回一个特殊值表示无人获胜。

下面的伪代码描述了这一算法。

```
1: function MAJORITY(A)
2:     M ← empty map
3:     for ∀a ∈ A do
4:         PUT(M, a, 1+ GET(M, a))
5:     max ← 0, m ← NIL
6:     for ∀(k, v) ∈ M do
7:         if max < v then
8:             max ← v, m ← k
9:     if max > |A|50% then
10:        return m
11:    else
12:        fail
```

对于 m 名候选人和 n 张选票，若使用自平衡树实现的 map（如红黑树 map），这一程序首先需要 $O(n \log m)$ 时间来构建 map；若使用散列表实现的 map，则所用时间为 $O(n)$。但是散列表所用的空间会更多。接下来，程序需要 $O(m)$ 的时间来遍历 map，然后寻找票数最多的候选人。表14.2给出了使用不同种类 map 所需的时间和空间的对比。

map	时间	空间
自平衡树	$O(n \log m)$	$O(m)$
散列	$O(n)$	最少 $O(m)$

表 14.2: 不同种类 map 的性能对比

Boyer 和 Moore 在 1980 年发现了一种巧妙的方法，如果存在超过半数的元素，可以只扫描一遍就找到它。并且这一方法只需要 $O(1)$ 的空间 [83]。

首先我们记录第一张选票投给的候选人为目前的获胜者，所赢得票数为 1。在接下来的扫描中，若下一张选票还投给目前的获胜者，就将获胜者的票数加 1；否则，下一张选票没有投给目前的获胜者，我们将获胜者的赢得的票数减 1。若获胜者的净赢得的票数变为 0，说明他不再是获胜者了，我们选择下一张选票上的候选人作为新的获胜者，并继续重复这一扫描过程。

假设选票的序列为：A, B, C, B, B, C, A, B, A, B, B, D, B。表14.3给出了这一扫描处理的各个步骤。

这里关键的一点是：若存在一个超过 50% 的众数，则它不可能被其它元素超越落选。但是，如果没有任何候选者赢得半数以上的选票，则最后所记录的“获胜者”

获胜者	净赢票数	扫描位置
A	1	A, B, C, B, B, C, A, B, A, B, B, D, B
A	0	A, B, C, B, B, C, A, B, A, B, B, D, B
C	1	A, B, C, B, B, C, A, B, A, B, B, D, B
C	0	A, B, C, B, B, C, A, B, A, B, B, D, B
B	1	A, B, C, B, B, C, A, B, A, B, B, D, B
B	0	A, B, C, B, B, C, A, B, A, B, B, D, B
A	1	A, B, C, B, B, C, A, B, A, B, B, D, B
A	0	A, B, C, B, B, C, A, B, A, B, B, D, B
A	1	A, B, C, B, B, C, A, B, A, B, B, D, B
A	0	A, B, C, B, B, C, A, B, A, B, B, D, B
B	1	A, B, C, B, B, C, A, B, A, B, B, D, B
B	0	A, B, C, B, B, C, A, B, A, B, B, D, B
B	1	A, B, C, B, B, C, A, B, A, B, B, D, B

表 14.3: 扫描选票的处理步骤

并无意义。此时需要再进行一轮扫描进行验证。

下面的算法实现了这一思路。

```
1: function MAJORITY(A)
2:     c ← 0
3:     for i ← 1 to |A| do
4:         if c = 0 then
5:             x ← A[i]
6:         if A[i] = x then
7:             c ← c + 1
8:         else
9:             c ← c − 1
10:    return x
```

若存在众数，这一算法首先扫描所有的选票。每扫描一张票，它根据此选票是支持还是反对当前的结果来增减获胜者的净赢票数。若净赢票数变为 0，表明当前的获胜者已落选，算法记录下一张选票投给的候选人为新的获胜者，并继续扫描。

这一过程是线性时间 $O(n)$ 的，所用空间仅仅是两个变量。一个用以记录当前的获胜者，另一个记录净赢得的票数。

当众数存在时，虽然上述算法可以将它找出。但当不存在众数时，这一算法仍会输出一个不正确的结果。下面的改进通过增加一轮扫描来进行验证。

```
1: function MAJORITY(A)
```

```
    c ← 0
    for i ← 1 to |A| do
        if c = 0 then
            x ← A[i]
        if A[i] = x then
            c ← c + 1
        else
            c ← c − 1
    c ← 0
    for i ← 1 to |A| do
        if A[i] = x then
            c ← c + 1
    if c > %50|A| then
        return x
    else
        fail
```

即使增加了验证的过程，这一算法的时间复杂度仍按为 $O(n)$，并且所用空间为常数。下面的 C++ 例子程序实现了这一算法[6]。

```
template<typename T>
T majority(const T* xs, int n, T fail) {
    T m;
    int i, c;
    for (i = 0, c = 0; i < n; ++i) {
        if (!c)
            m = xs[i];
        c += xs[i] == m ? 1 : -1;
    }
    for (i = 0, c = 0; i < n; ++i, c += xs[i] == m);
    return c * 2 > n ? m : fail;
}
```

Boyer-Moore 众数算法也可以用纯函数的方式实现。我们不再使用变量来记录和更新信息,而是使用累积器(accumulator)的方法。定义核心算法的函数为 $maj(c, n, L)$，它接受一个选票列表 L，目前的获胜者 c，和净赢得的票数 n。若选票列表不为空，则 c 在开始的时候为第一张选票的结果 l_1，净赢得的票数为 1，即 $maj(l_1, 1, L')$，其中 L' 是除 l_1 以外的剩余选票。下面是这个函数的详细定义：

$$maj(c, n, L) = \begin{cases} c & : & L = \phi \\ maj(c, n+1, L') & : & l_1 = c \\ maj(l_1, 1, L') & : & n = 0 \wedge l_1 \neq c \\ maj(c, n-1, L') & : & otherwise \end{cases} \tag{14.19}$$

[6]这是一个更加类似 C 语言例子，我们只是使用了 C++ 的模板来抽象元素的类型。

我们还需要定义一个函数来验证所得的结果是否超过半数。最终的算法首先检查选票列表，若为空，则不存在众数，否则它通过 Boyer-Moore 算法找到一个结果 c，然后再扫描一遍选票列表计算 c 总共赢得的选票是否过半。

$$majority(L) = \begin{cases} fail & : & L = \phi \\ c & : & c = maj(l_1, 1, L'), |\{x | x \in L, x = c\}| > \%50|L| \\ fail & : & otherwise \end{cases} \tag{14.20}$$

下面的 Haskell 例子程序实现了这一算法。

```
majority :: (Eq a) ⇒ [a] → Maybe a
majority [] = Nothing
majority (x:xs) = let m = maj x 1 xs in verify m (x:xs)

maj c n [] = c
maj c n (x:xs) | c == x = maj c (n+1) xs
               | n == 0 = maj x 1 xs
               | otherwise = maj c (n-1) xs

verify m xs = if 2 * (length $ filter (==m) xs) > length xs
              then Just m else Nothing
```

最大子序列和

Jon Bentley 给出过另一个类似的趣题 [2]。给定一个序列，如何找出它的子序列和的最大值？例如，下表所示的序列中，子序列 {19, -12, 1, 9, 18} 的和最大，为 35。

3	-13	19	-12	1	9	18	-16	15	-15

这里，我们只要找出最大和的值。如果所有元素都是正数，显然答案就是全部元素的和。另外一个特殊情况是所有元素都是负数。我们定义空序列的最大和是 0。

最简单的方法是穷举：计算出所有子序列的和，然后挑选最大的作为答案。这一方法的复杂度为平方级别。

```
1: function MAX-SUM(A)
2:     m ← 0
3:     for i ← 1 to |A| do
4:         s ← 0
5:         for j ← i to |A| do
6:             s ← s + A[j]
7:             m ← MAX(m, s)
8:     return m
```

穷举法没有复用任何此前已经计算出的结果。借鉴 Boyer-Moore 众数算法的思路，我们可以一边扫描，一边记录下以当前位置结尾的子序列的最大和。同时我们还需要记录下目前为止所找到的最大和，图14.11给出了扫描时所保持的不变性质。

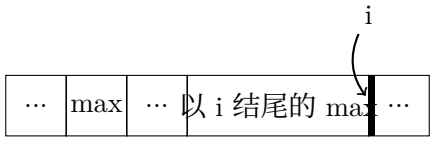


图 14.11: 扫描时的不变性质

在任何时候，当我们扫描到第 i 个位置时，目前找到的最大和记为 A。同时，我们记录下以 i 结尾的子序列的最大和为 B。A 和 B 并不一定相等，实际上，我们总保持 $B \leq A$ 的关系。当 B 和下一个元素相加，从而超过 A 时，我们就用这个更大的结果替换 A。当 B 加上下一个元素后，变为负数时，我们将 B 重新设置为 0。下表给出了扫描处理序列 $\{3, -13, 19, -12, 1, 9, 18, -16, 15, -15\}$ 时的各个步骤。

最大和	以 i 结尾的子序列最大和	尚未扫描的部分
0	0	$\{3, -13, 19, -12, 1, 9, 18, -16, 15, -15\}$
3	3	$\{-13, 19, -12, 1, 9, 18, -16, 15, -15\}$
3	0	$\{19, -12, 1, 9, 18, -16, 15, -15\}$
19	19	$\{-12, 1, 9, 18, -16, 15, -15\}$
19	7	$\{1, 9, 18, -16, 15, -15\}$
19	8	$\{9, 18, -16, 15, -15\}$
19	17	$\{18, -16, 15, -15\}$
35	35	$\{-16, 15, -15\}$
35	19	$\{15, -15\}$
35	34	$\{-15\}$
35	19	$\{\}$

表 14.4: 扫描序列求最大子序列和的步骤

这一算法可以描述如下：

```
1: function MAX-SUM(V)
2:     A ← 0, B ← 0
3:     for i ← 1 to |V| do
4:         B ← MAX(B + V[i], 0)
5:         A ← MAX(A, B)
```

也可以用函数式的方式实现这一算法。我们不再更新变量 A 和 B，而是把它们作为尾递归的累积器。为了找到序列 L 的最大子序列和，我们调用函数 $max_{sum}(0, 0, L)$。

$$max_{sum}(A, B, L) = \begin{cases} A & : \quad L = \phi \\ max_{sum}(A', B', L') & : \quad otherwise \end{cases} \tag{14.21}$$

其中

$$B' = max(l_1 + B, 0)$$
$$A' = max(A, B')$$

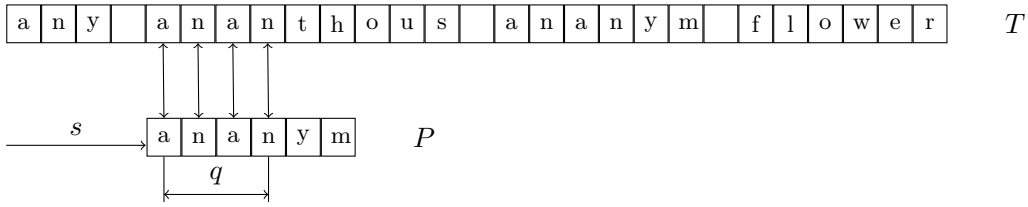
下面的 Haskell 例子程序实现了这一算法。

```
maxsum = msum 0 0 where
  msum a _ [] = a
  msum a b (x:xs) = let b' = max (x+b) 0
                        a' = max a b'
                    in msum a' b' xs
```

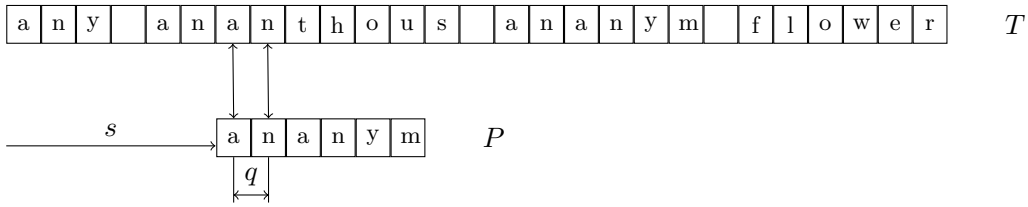
KMP

字符串搜索是一类很重要的问题。所有的文本编辑器软件都带有字符串搜索功能。在 Trie、Patricia 和后缀树章节，我们介绍了一些字符串搜索常用的数据结构。本节中，我们介绍两种利用信息复用进行字符串搜索的算法。

有些编程环境提供了内置的字符串搜索工具，但是大多数是用暴力解法，包括 ANSI C 标准库中的 `strstr` 函数，C++ 标准模板库中的 `find`，以及 Java 标准库 JDK 中的 `indexOf`。图14.12描述了逐一比较字符的过程。



(a) 偏移量 $s = 4$，接着，连续有 $q = 4$ 个字符相同，但是第 5 个字符不同。



(b) 比较的起始位置移动到 $s = 4 + 2 = 6$。

图 14.12: 在文本“any ananthous ananym flower”中寻找“ananym”

考虑我们在文本 T 中搜索字符串 P，如图14.12 (a) 所示，在偏移量为 $s = 4$ 时，处理过程逐一检查 P 和 T 中的字符是否相等。前 4 个字符都是 anan，但是第 5 个字符在 P 中是 y，而在文本 T 中是 t，它们不相等。

此时，逐一比较过程立即终止，我们将 s 加 1，也就是把 P 向右移动 1 个位置，然后重新比较 ananym 和 nantho……实际上，s 的增量可以超过 1。这是因为，当我们发现第 5 个字符不等的时候，已经比较过前 4 面个字符 anan 了。并且最前面的两

个字符 an 恰好是 anan 的后缀。因此更有效的做法是将 s 增加 2，也就是把 P 向右移动两个位置，如图14.12 (b) 所示。这样，我们就复用了前面已经比较过的 4 个字符的信息，从而跳过大量无需比较的位置。

Knuth、Morris 和 Pratt 根据这一思路给出了一个高效的字符串匹配算法 [85]，人们把三位作者的名字合在一起，称作 KMP 算法。

简洁起见，我们记文本 T 中的前 k 个字符组成的串为 T_k，即 T_k 为文本 T 的 k 个字符前缀。

为了把 P 高效向右移动 s 个位置，我们需要定义一个关于 q 的函数，其中 q 是成功匹配的字符个数。例如在图14.12 (a) 中，q 的值为 4，即第 5 个字符不匹配。

什么情况下向右移动的距离 s 可以大于 1 呢？如图14.13所示，若可以将 P 向右移动，则一定存在某个 k，使得 P 中的前 k 个字符和前缀 P_q 的最后 k 个字符相同。也就是说，前缀 P_k 同时是 P_q 的后缀。

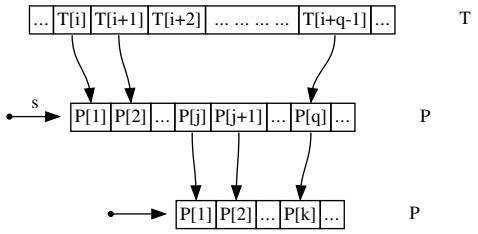


图 14.13: P_k 既是 P_q 的前缀，也是 P_q 的后缀

当然有可能不存在同时也是后缀的前缀。如果我们认为空串同时是任何其他字符串的前缀和后缀，则总存在一个解 $k = 0$。如果存在多个 k 满足，为了避免漏掉任何可能的候选位置，我们需要找到同时既是前缀又是后缀的最大的 k。我们定义一个前缀函数$\pi(q)$，它告诉我们当第 $q + 1$ 个字符不匹配时应该回退的位置 [4]。

$$\pi(q) = max\{k | k < q \wedge P_k \sqsupset P_q\} \tag{14.22}$$

其中，$A \sqsupset B$ 表示“A 是 B 的后缀”。这一函数的使用方法如下：当我们在文本 T 中，以 offset 为 s 尝试匹配 P 时，若前 q 个字符都相同，而接下来的字符不同，我们接下来通过 $\pi(q)$ 找到一个回退的位置 q'，然后重新尝试比较 $P[q']$ 和文本中的字符。根据这一思路，KMP 的核心算法可以描述如下：

```
1: function KMP(T, P)
2:     n ← |T|, m ← |P|
3:     build prefix function π from P
4:     q ← 0                                  ▷ 记录已经匹配的字符个数
5:     for i ← 1 to n do
6:         while q > 0 ∧ P[q + 1] ≠ T[i] do
7:             q ← π(q)
```

```
        if P[q + 1] = T[i] then
            q ← q + 1
        if q = m then
            found one solution at i − m
            q ← π(q)                                  ▷ 继续寻找下一个可能的位置
```

虽然式 (14.22) 给出了前缀函数 $\pi(q)$ 的定义，但是简单寻找最长后缀的效率很低。实际上，我们可以进一步复用信息，来快速构造前缀函数。

最简单的情况是第一个字符就不相等。这种情况下，最长的前缀，同时也是后缀显然是空串，因此 $\pi(1) = k = 0$。记最长的前缀为 P_k。此时，$P_k = P_0$ 等于空串。

此后，当我们扫描到 P 中的第 q 个字符时，我们总有，前缀函数的所有值 $\pi(i)$，i 在 $\{1, 2, ..., q-1\}$ 都已经算出并记录下来了，并且目前最长的前缀 P_k 同时也是 P_{q-1} 的后缀。如图14.14所示，若 $P[q] = P[k+1]$，则我们找到了一个更大的 k，我们将 k 的最大值加一；否则，若两个字符不等，我们使用 $\pi(k)$ 回退到一个较短的 $P_{k'}$，其中 $k' = \pi(k)$，然后比较这个新前缀的下一个字符是否和第 q 个字符相等。我们需要重复这一步骤，直到 k 变成 0（表示只有空串满足条件），或者和第 q 个字符相等。

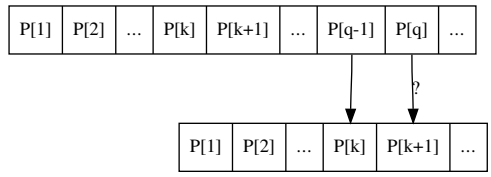


图 14.14: P_k 是 P_{q-1} 的后缀，比较 $P[q]$ 和 $P[k+1]$

KMP 算法中，构建前缀函数的过程可以描述如下：

```
function BUILD-PREFIX-FUNCTION(P)
    m ← |P|, k ← 0
    π(1) ← 0
    for q ← 2 to m do
        while k > 0 ∧ P[q] ≠ P[k + 1] do
            k ← π(k)
        if P[q] = P[k + 1] then
            k ← k + 1
        π(q) ← k
    return π
```

下表列出了为字符串“ananym”构建前缀函数的步骤。表中的 k 实际上表示满足式 (14.22) 的最大 k。

下面的 Python 例子程序实现了完整的 KMP 算法。

```
def kmp_match(w, p):
```

q	P_q	k	P_k
1	a	0	“”
2	an	0	“”
3	ana	1	a
4	anan	2	an
5	anany	0	“”
6	ananym	0	“”

表 14.5: 构建前缀函数的步骤

```
    n = len(w)
    m = len(p)
    fallback = fprefix(p)
    k = 0 # how many elements have been matched so far.
    res = []
    for i in range(n):
        while k > 0 and p[k] ≠ w[i]:
            k = fallback[k] #fall back
        if p[k] == w[i]:
            k = k + 1
        if k == m:
            res.append(i+1-m)
            k = fallback[k-1] # look for next
    return res

def fprefix(p):
    m = len(p)
    t = [0]*m  # fallback table
    k = 0
    for i in range(2, m):
        while k>0 and p[i-1] ≠ p[k]:
            k = t[k-1] #fallback
        if p[i-1] == p[k]:
            k = k + 1
        t[i] = k
    return t
```

KMP 算法相当于在搜索前将待搜索的字符串进行预处理。因此它可以最大程度地复用已知的匹配结果。

构建前缀函数的分摊性能为 $O(m)$，可以使用势能分析法证明 [4]。使用同样的方法可以进一步证明搜索算法本身的性能也是线性时间的。因此总体时间性能为 $O(m+n)$，同时需要额外的 $O(m)$ 空间来记录前缀函数的表格。

如果不仔细分析，可能会认为不同形式的待搜索字符串会影响 KMP 的性能。考虑在一个长度为 n 个 a 的文本 “aaa...a” 中，搜索长度为 m 个 a 的字符串 “aaa...a”。因为所有的字符都相同，当最后一个字符匹配完成后，我们只能回退一个字符，并且此后不断重复回退 1 个字符。即使在这种极端情况下，KMP 算法依旧是线性时间的

(为什么?)。请尝试考虑更多情况，例如 $P = aaaa...b$，$T = aaaa...a$，并分析 KMP 的性能。

纯函数式 KMP 算法

用纯函数式的方法实现 KMP 算法会比较困难。命令式的 KMP 算法大量使用数组来保存前缀函数的值。虽然可以使用纯函数式的序列数据结构来代替数组，但序列通常使用手指树来实现。与命令式环境中的数组相比，手指树随机访问元素的性能为对数时间[7]。

Richard Bird 给出了一个使用 fold fusion 定理推导 KMP 算法的过程（[1] 第 17 章)。本节中，我们首先给出一个暴力法的前缀函数构造方法，然后逐步改进得到 KMP 算法。

在函数式环境中，文本和待搜索的字符串本质上都是用单向链表表示的列表。在扫描过程中，两个列表被分解，每个列表都被分成两部分。如图14.15所示，待搜索的字符串的前 j 个字符都相附，接下来要比较 $T[i+1]$ 和 $P[j+1]$。如果相等，就将这一字符添加到已成功比较的部分。但是由于字符串由单向链表表示，向尾部添加的性能和其长度 j 成比例。

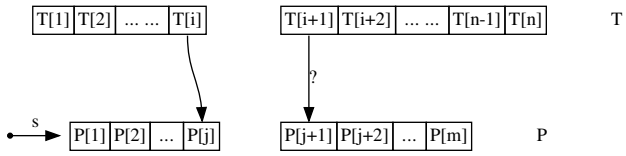


图 14.15: P 的前 j 个字符都相符，接下来比较 $P[j+1]$ 和 $T[i+1]$

记文本的前 i 个字符为 T_p，表示 T 的前缀，剩余的字符为 T_s，表示 T 的后缀；同样，记 P 的前 j 个字符为 P_p，剩余字符为 P_s；记 T_s 中的第一个字符为 t，P_s 中的第一个字符为 p。我们可以得到如下的“cons”关系。

$$\begin{aligned} T_s &= cons(t, T'_s) \\ P_s &= cons(p, P'_s) \end{aligned} \tag{14.23}$$

若 $t = p$，则下面的更新过程需要线性时间：

$$\begin{aligned} T'_p &= T_p \cup \{t\} \\ P'_p &= P_p \cup \{p\} \end{aligned} \tag{14.24}$$

我们在队列一章中曾经介绍过一种方法，可以解决这一问题。通过使用一对列表，一个 front 列表和一个 rear 列表，可以将线性时间的添加操作转换成常数时间的链

[7]我们在这里不使用数组。虽然在某些函数式编程环境中，例如 Haskell 提供了可以在常数时间进行随机访问的数组。

接操作。为此，需要将前缀的部分用逆序表达。

$$\begin{aligned} T &= T_p \cup T_s = reverse(reverse(T_p)) \cup T_s = reverse(\overleftarrow{T_p}) \cup T_s \\ P &= P_p \cup P_s = reverse(reverse(P_p)) \cup P_s = reverse(\overleftarrow{P_p}) \cup P_s \end{aligned} \tag{14.25}$$

我们分别用 $(\overleftarrow{T_p}, T_s)$ 和 $(\overleftarrow{P_p}, P_s)$ 来表达文本和待搜索的字符串。这样当 $t = p$ 时，就可以用常数时间，快速更新前缀部分。

$$\begin{aligned} \overleftarrow{T'_p} &= cons(t, \overleftarrow{T_p}) \\ \overleftarrow{P'_p} &= cons(p, \overleftarrow{P_p}) \end{aligned} \tag{14.26}$$

KMP 查找算法开始时，已成功匹配的前缀部分初始化为空串。

$$search(P, T) = kmp(\pi, (\phi, P)(\phi, T)) \tag{14.27}$$

其中 π 是此前介绍过的前缀函数。除构造前缀函数外的 KMP 核心算法可以定义如下。

$$kmp(\pi, (\overleftarrow{P_p}, P_s), (\overleftarrow{T_p}, T_s)) = \begin{cases} \{|\overleftarrow{T_p}|\} & : & P_s = \phi \wedge T_s = \phi \\ \phi & : & P_s \neq \phi \wedge T_s = \phi \\ \{|\overleftarrow{T_p}\} \cup kmp(\pi, \pi(\overleftarrow{P_p}, P_s), (\overleftarrow{T_p}, T_s)) & : & P_s = \phi \wedge T_s \neq \phi \\ kmp(\pi, (\overleftarrow{P'_p}, P'_s), (\overleftarrow{T'_p}, T'_s)) & : & t = p \\ kmp(\pi, \pi(\overleftarrow{P_p}, P_s), (\overleftarrow{T'_p}, T'_s)) & : & t \neq p \wedge \overleftarrow{P_p} = \phi \\ kmp(\pi, \pi(\overleftarrow{P_p}, P_s), (\overleftarrow{T_p}, T_s)) & : & t \neq p \wedge \overleftarrow{P_p} \neq \phi \end{cases} \tag{14.28}$$

第一行表示，若同时扫描完文本和待搜索字串，则获得一个解，同时算法结束。这里我们使用文本中的右侧位置作为搜索到的位置。如果要使用左侧位置，只需要用右侧位置减去待搜索串的长度即可。简单起见，在函数式的解法中，我们使用右侧位置。

第二行表示，若文本已经扫描结束，但是待搜索的字串中仍然有尚未扫描的字符，则不存在解，并且算法结束。

第三行表示，如果带搜索的字串已全部扫描匹配成功，但是文本中仍然存在未扫描的字符，我们得到一个解，但是需要继续搜索。此时我们调用前缀函数 π，获得下一个继续搜索的起始位置。

第四行处理待搜索字串中的下一个字符和文本中的下一个字符相同的情况。此时需要同时向前移动一个字符，然后递归进行搜索。

如果下一个字符不同，并且恰好是待搜索字串的第一个字符，我们只需要移动到文本中的下一个字符，然后重新查找。否则，如果不是待搜索字串中的第一个字符，我们就调用前缀函数 π，获取到回退的位置，以继续进行搜索。

可以用暴力方法构造前缀函数，只要简单地按照式 (14.22) 的定义即可，如 (14.29) 所示。

$$\pi(\overleftarrow{P_p}, P_s) = (\overleftarrow{P'_p}, P'_s) \tag{14.29}$$

其中

$$\begin{aligned} P'_p &= longest(\{s | s \in prefixes(P_p), s \sqsupset P_p\}) \\ P'_s &= P - P'_p \end{aligned} \tag{14.30}$$

每次计算回退的位置时，暴力法都简单地穷举所有 P_p 的前缀，检查它是否同时也是 P_p 的后缀，然后选择最长的一个作为结果。这里我们使用了减号表示获取列表的不同部分。

这里需要避免一种特殊情况。由于任何字符串本身都同时是自己的前缀和后缀，即 $P_p \sqsubset P_p$ 和 $P_p \sqsupset P_p$ 总成立，因此不能将 P_p 作为一个候选的前缀。下面给出了穷举前缀算法的定义：

$$prefixes(L) = \begin{cases} \{\phi\} & : \quad L = \phi \vee |L| = 1 \\ cons(\phi, map(\lambda_s \cdot cons(l_1, s), prefixes(L'))) & : \quad otherwise \end{cases} \tag{14.31}$$

下面的 Haskell 例子程序实现了对应的字符串查找算法。

```
kmpSearch1 ptn text = kmpSearch' next ([], ptn) ([], text)

kmpSearch' _ (sp, []) (sw, []) = [length sw]
kmpSearch' _ _ (_, []) = []
kmpSearch' f (sp, []) (sw, ws) = length sw : kmpSearch' f (f sp []) (sw, ws)
kmpSearch' f (sp, (p:ps)) (sw, (w:ws))
    | p == w = kmpSearch' f ((p:sp), ps) ((w:sw), ws)
    | otherwise = if sp ==[] then kmpSearch' f (sp, (p:ps)) ((w:sw), ws)
                  else kmpSearch' f (f sp (p:ps)) (sw, (w:ws))

next sp ps = (sp', ps') where
    prev = reverse sp
    prefix = longest [xs | xs ← inits prev, xs `isSuffixOf` prev]
    sp' = reverse prefix
    ps' = (prev ++ ps) \\ prefix
    longest = maximumBy (compare `on` length)

inits [] = [[]]
inits [_] = [[]]
inits (x:xs) = [] : (map (x:) $ inits xs)
```

这一算法不仅性能差，而且也很复杂。我们可以对其略作简化。观察 KMP 搜索过程，它实际上是一个从左向右的对文本进行扫描的过程，因此可以使用 fold 来表示（参见附录 A）。首先，在 fold 的过程中，我们可以让每一个字符对应一个索引，如下：

$$zip(T, \{1, 2, ...\}) \tag{14.32}$$

将文本和自然数 zip 起来，得到一个列表，每个元素都是一对值。例如文本“The quick brown fox jumps over the lazy dog”这样处理后的结果是：T, 1), (h, 2), (e, 3), ..., (o, 42), (g, 43)。

fold 起始时的状态包含两部分，第一部分是待搜索字符串 (P_p, P_s)，其中前缀起始时为空，后缀为完成的待搜索串，即 (ϕ, P)。为了方便，我们暂时不用 $(\overleftarrow{P_p}, P_s)$ 的表示法，在最终的定义中我们需要再次改回来。起始状态的另外一部分是已找到的解的列表，它初始为空。fold 结束时，这一列表包含所有找到的解。我们需要将其取出，作为最终的结果。这样核心的 KMP 算法定义可简化如下：

$$kmp(P, T) = snd(fold(search, ((\phi, P), \phi), zip(T, \{1, 2, ...\}))) \tag{14.33}$$

这里唯一的“黑盒子”是 $search$ 函数，它接受一个状态，和一个字符——索引对，计算后返回一个新的状态作为结果。记 P_s 中的第一个字符为 p，剩余的字符为 P'_s（即 $P_s = cons(p, P'_s)$），我们有如下的定义：

$$search(((P_p, P_s), L), (c, i)) = \begin{cases} ((P_p \cup p, P'_s), L \cup \{i\}) & : \quad p = c \wedge P'_s = \phi \\ ((P_p \cup p, P'_s), L) & : \quad p = c \wedge P'_s \neq \phi \\ ((P_p, P_s), L) & : \quad P_p = \phi \\ search((\pi(P_p, P_s), L), (c, i)) & : \quad otherwise \end{cases} \tag{14.34}$$

如果 P_s 中的第一个字符和当前扫描的字符 c 相等，我们需要检查是否待搜索串种的所有字符都已扫描完毕，如果已完毕，则找到了一个解，我们将这一位置 i 记录到列表 L 中；如果尚未完毕，我们向前移动一个字符，然后继续。如果 p 和 c 不同，我们需要回退到某个位置，然后重新搜索。但是有一个特殊情况，我们不能回退：当 P_p 为空时，我们需要保持当前的状态。

前缀函数 π 的定义也可以略微简化。我们要寻找的是一个最长子串，它即是 P_p 前缀，同时也是它后缀。我们可以从右向左扫描。对于任何非空列表 L，记表中第一个元素为 l_1，剩余的部分为 L'，定义函数 $init(L)$ 返回除最后一个元素外的所有其他元素。

$$init(L) = \begin{cases} \phi & : \quad |L| = 1 \\ cons(l_1, init(L')) & : \quad otherwise \end{cases} \tag{14.35}$$

注意，这一定义不能处理列表为空的情况。从右向左扫描 P_p，就是首先检查 $init(P_p) \sqsupset P_p$ 是否成立，如果是，则成功；否则我们接下来检查 $init(init(P_p))$ 是否可以，并且重复这一过程直到最左侧。这样前缀函数的定义就可以简化如下：

$$\pi(P_p, P_s) = \begin{cases} (P_p, P_s) & : \quad P_p = \phi \\ fallback(init(P_p), cons(last(P_p), P_s)) & : \quad otherwise \end{cases} \tag{14.36}$$

其中

$$fallback(A, B) = \begin{cases} (A, B) & : \quad A \sqsupset P_p \\ (init(A), cons(last(A), B)) & : \quad otherwise \end{cases} \tag{14.37}$$

由于空串是任何字符串的后缀，因此函数 fallback 一定能结束。函数 $last(L)$ 返回一个列表的最后一个元素，它同样是一个线性时间的操作（参见附录 A）。但如果

我们使用 $\overleftarrow{P_p}$ 的表示法，则获取最后一个元素就是一个常数时间的操作。这一改进的前缀函数的复杂度为线性时间，但和命令式的算法相比，仍然很慢。因为命令式算法可以在常数时间进行前缀函数的检索。下面的 Haskell 例子程序实现了这一改进。

```
failure ([], ys) = ([], ys)
failure (xs, ys) = fallback (init xs) (last xs:ys) where
    fallback as bs | as `isSuffixOf` xs = (as, bs)
                   | otherwise = fallback (init as) (last as:bs)

kmpSearch ws txt = snd $ foldl f (([], ws), []) (zip txt [1..]) where
    f (p@(xs, (y:ys)), ns) (x, n) | x == y = if ys==[] then ((xs++[y], ys), ns++[n])
                                             else ((xs++[y], ys), ns)
                                  | xs == [] = (p, ns)
                                  | otherwise = f (failure p, ns) (x, n)
    f (p, ns) e = f (failure p, ns) e
```

瓶颈在于，在纯函数式的环境中，我们无法使用内置的 array 来记录前缀函数。实际上，前缀函数可以被看作是一个状态转移函数。它根据字符匹配成功与否将一个状态转移到另一个状态。我们可以将这样的状态转换抽象为一棵树。在支持代数数据类型（algebraic data type）的环境中，例如 Haskell，这样的状态树可以定义如下：

```
data State a = E | S a (State a) (State a)
```

一个状态或者为空，或者包含三部分：当前的状态，如果匹配失败后转移到的状态，和匹配成功后转移到的状态。这一定义和二叉树的定义很像。我们将其称为“左侧失败，右侧成功”树。这里的具体状态为 (P_p, P_s)。

在命令式的 KMP 算法中，我们通过待搜索字串构造前缀函数。与此类似，我们可以通过待搜索字串构造状态转移树。我们从起始状态 (ϕ, P) 开始，它的两个子状态最初为空。我们调用上面定义的 π 获得一个新状态，用它替换掉左侧子节点，然后通过向前前进一个字符得到一个新状态并替换右侧子节点。这里有一种特殊情况，当状态转移到 (P, ϕ) 时，如果匹配成功，我们无法继续前进。这样的节点只含有失败的子状态。下面定义了构造状态转移树的的函数。

$$build((P_p, P_s), \phi, \phi) = \begin{cases} build(\pi(P_p, P_s), \phi, \phi) & : \quad P_s = \phi \\ build((P_p, P_s), L, R) & : \quad otherwise \end{cases} \tag{14.38}$$

其中

$$\begin{aligned} L &= build(\pi(P_p, P_s), \phi, \phi) \\ R &= build((P_s \cup \{p\}, P'_s), \phi, \phi)) \end{aligned}$$

其中 p 和 P'_s 的含义和此前相同，p 是 P_s 中的第一个字符，P'_s 是剩余部分。最有趣的一点是，build 函数永远不会结束。它无休无止地构造一棵无穷树。在严格的（strict）编程环境中，调用这样的函数会陷入麻烦。但在支持惰性求值的环境中，只有被使用的节点才会被构造。Lisp 方言 Scheme 和 Haskell 都可以构造这样的无穷状态树。在命令式环境中，我们通常使用指向祖先节点的指针来实现无穷状态树。

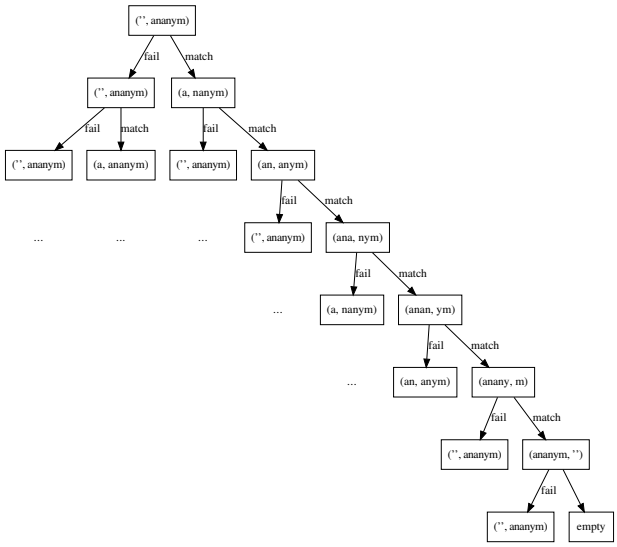


图 14.16: 从字符串“ananym”构造的无穷状态树

图14.16描述了从字符串“ananym”对应的无穷状态树。其中最右侧的边对应了字符匹配一直连续成功的特殊情况。此后，由于所有的字符都匹配完毕，所有后继的右侧子树为空。根据这一点，我们可以定义一个辅助函数来判断是否一个状态代表待搜索字符串已经完全匹配成功。

$$match((P_p, P_s), L, R) = \begin{cases} True & : & P_s = \phi \\ False & : & otherwise \end{cases} \tag{14.39}$$

通过使用状态转移树，我们可以用一个自动机来实现 KMP 算法。

$$kmp(P, T) = snd(fold(search, (Tr, []), zip(T, \{1, 2, ...\}))) \tag{14.40}$$

其中，$Tr = build((\phi, P), \phi, \phi)$ 是无穷状态转移树。函数 $search$ 根据匹配成功与否，使用这棵树进行状态转移。记 P_s 中的第一个字符为 p，剩余部分为 P'_s，A 代表已找到的解的位置。

$$search((((P_p, P_s), L, R), A), (c, i)) = \begin{cases} (R, A \cup \{i\}) & : & p = c \wedge match(R) \\ (R, A) & : & p = c \wedge \neg match(R) \\ ((((P_p, P_s), L, R), A) & : & P_p = \phi \\ search((L, A), (c, i)) & : & otherwise \end{cases} \tag{14.41}$$

下面的 Haskell 例子程序实现了这一算法。

```
data State a = E | S a (State a) (State a) -- state, ok-state, fail-state
             deriving (Eq, Show)

build :: (Eq a)⇒State ([a], [a]) → State ([a], [a])
build (S s@(xs, []) E E) = S s (build (S (failure s) E E)) E
```

```
build (S s@(xs, (y:ys)) E E) = S s l r where
    l = build (S (failure s) E E) -- fail state
    r = build (S (xs++[y], ys) E E)

matched (S (_, []) _ _) = True
matched _ = False

kmpSearch3 :: (Eq a) => [a] -> [a] -> [Int]
kmpSearch3 ws txt = snd $ foldl f (auto, []) (zip txt [1..]) where
    auto = build (S ([], ws) E E)
    f (s@(S (xs, ys) l r), ns) (x, n)
        | [x] `isPrefixOf` ys = if matched r then (r, ns++[n])
                                else (r, ns)
        | xs == [] = (s, ns)
        | otherwise = f (l, ns) (x, n)
```

目前的瓶颈在于构造状态转移树的时候，需要调用 π 函数计算回退的位置，而前缀函数 π 的定义效率很差。这是由于它每次都从右向左穷举所有可能前缀。

由于状态树是无穷的，我们可以使用处理无穷数据结构的常见方法。典型的例子就是斐波那契数列。斐波那契数列的前两个值为 0 和 1，其余的斐波那契数可以通过将前面的两个值相加得到：

$$\begin{aligned} F_0 &= 0 \\ F_1 &= 1 \\ F_n &= F_{n-1} + F_{n-2} \end{aligned} \tag{14.42}$$

这样，就可以依次列出所有的斐波那契数。

$$\begin{aligned} F_0 &= 0 \\ F_1 &= 1 \\ F_2 &= F_1 + F_0 \\ F_3 &= F_2 + F_1 \\ &... \end{aligned} \tag{14.43}$$

将上述等式左右两侧的所有数字汇集起来,定义无穷斐波那契数列为 $F = \{0, 1, F_1, F_2, ...\}$，我们有下面的等式：

$$\begin{aligned} F = & \{0, 1, F_1 + F_0, F_2 + F_1, ...\} \\ = & \{0, 1\} \cup \{x + y | x \in \{F_0, F_1, F_2, ...\}, y \in \{F_1, F_2, F_3, ...\}\} \\ = & \{0, 1\} \cup \{x + y | x \in F, y \in F'\} \end{aligned} \tag{14.44}$$

其中 $F' = tail(F)$ 是除第一个元素外的所有斐波那契数。在支持惰性求值的环境中，如 Haskell，这一定义可以实现如下。

```
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)
```

无穷斐波那契数列的递归定义可以启发我们找到避免使用函数 π 进行回退的方法。记状态转移树为 T，我们可以定义一个用这棵树匹配字符时的状态转移函数。

$$trans(T, c) = \begin{cases} root & : & T = \phi \\ R & : & T = ((P_p, P_s), L, R), c = p \\ trans(L, c) & : & otherwise \end{cases} \tag{14.45}$$

如果匹配一个字符时节点为空，我们转移到树的根节点。稍后我们会定义根节点。否则，我们比较字符 c 和 P_s 的第一个字符 p。如果相等，我们就转移到右侧子树表示成功；否则，我们转移到左侧子树表示失败。

通过使用状态转移函数，我们可以定义一个新的状态树构造算法。原理和前面的斐波那契序列类似。

$$build(T, (P_p, P_s)) = ((P_p, P_s), T, build(trans(T, p), (P_p \cup \{p\}, P'_s))) \tag{14.46}$$

等式右侧包含三部分。第一部分是我们正在搜索的状态 (P_p, P_s)；如果匹配失败，由于 T 本身可以处理任何失败的情况，我们直接使用它作为左侧子树；否则匹配成功，我们前进一个字符，递归构造右侧子树，并调用我们定义的状态转移函数。

这里还必须处理一种特殊情况，如果 P_s 为空，表示匹配了所有的字符，根据上面的定义，将不存在后继的右侧子树。综合起来，我们可以得到下面的构造函数。

$$build(T, (P_p, P_s)) = \begin{cases} ((P_p, P_s), T, \phi) & : & P_s = \phi \\ ((P_p, P_s), T, build(trans(T, p), (P_p \cup \{p\}, P'_s))) & : & otherwise \end{cases} \tag{14.47}$$

最后，我们还需要定义无穷状态转移树的根节点，用以初始化构造过程。

$$root = build(\phi, (\phi, P)) \tag{14.48}$$

使用这一根节点定义，我们可以给出一个新的 KMP 搜索算法。

$$kmp(P, T) = snd(fold(trans, (root, []), zip(T, \{1, 2, ...\}))) \tag{14.49}$$

下面的 Haskell 例子程序实现了这一 KMP 算法。

```
kmpSearch ws txt = snd $ foldl tr (root, []) (zip txt [1..]) where
    root = build' E ([], ws)
    build' fails (xs, []) = S (xs, []) fails E
    build' fails s@(xs, (y:ys)) = S s fails succs where
        succs = build' (fst (tr (fails, []) (y, 0))) (xs++[y], ys)
    tr (E, ns) _ = (root, ns)
    tr ((S (xs, ys) fails succs), ns) (x, n)
        | [x] `isPrefixOf` ys = if matched succs then (succs, ns++[n]) else (succs, ns)
        | otherwise = tr (fails, ns) (x, n)
```

图14.17给出了在文本“anal”中搜索“anaym”的前 4 步。由于前三步的字符都匹配成功,所以这 3 个状态的左侧子树都没有被构造。它们被标记为“?”。在第 4 步,字符匹配

失败,因此无需构造右侧的子树。同时,我们必须根据 $trans(right(right(right(T))), n)$ 的结果构造左侧的子树，其中函数 $right(T)$ 返回树 T 的右侧子树。根据构造过程和状态转移的定义，这一结果最终展开到一个具体的状态 $((a, nanym), L, R)$。具体的推导过程留给读者作为练习。

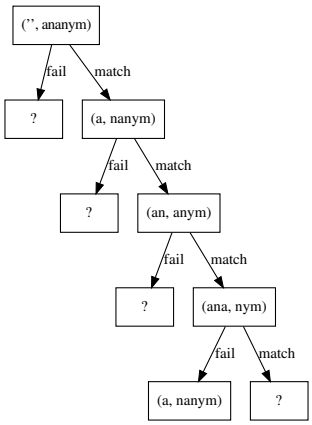


图 14.17: 在文本“anal”中搜索字符串“ananym”，按需构造构造状态转移树

这一算法的实现依赖于惰性求值。所有被转移到的状态都是按需构造。构造过程的分摊复杂度为 $O(m)$，算法的整体分摊性能为 $O(m+n)$。读者可以参考 [1] 了解详细的证明。

在我们此前介绍的很多形况中，函数式算法通常比较简洁。但是在 KMP 搜索中，命令式算法却更加简单、直观。这主要是由于我们通过无穷状态转移树来模拟内置数组造成的。

Boyer-Moore 字符串匹配算法

Boyer-Moore 字符串匹配算法是 1977 年发现的另一种高效的字符串查找方法 [86]。它的思想来自于下面的一些事实。

不良字符（bad-character）启发条件

当匹配待搜索的字符串时，即使从左边开始有若干字符都匹配成功，如果最后一个字符不相等，最终结果仍然失败，如图14.18所示。而且，即使我们将待搜索字符串向右侧平移 1 到 2 个单位，匹配仍然会失败。实际上，待查找的字符串“ananym”的长度为 6，最后一个字符是‘m’，但是文本中对应的字符是‘h’。它根本没有出现在待搜索的字符串中。我们据此可以直接向右侧平移 6 个单位。

从这点可以得到不良字符规则。我们可以对待搜索字符串进行预处理。如果文本中的字符集已知，我们可以找到所有不存在于待搜索串中的不良字符。在后继的扫描过程中，只要遇到不良字符，我们就可以立即向右侧移动一个待搜索串长度的距离。

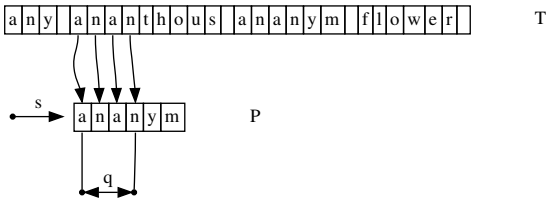
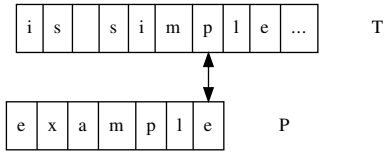
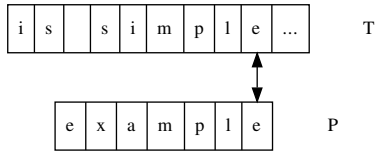


图 14.18: 因为字符‘h’没有出现在待搜索的字符串中，向右侧平移的距离如果小于 6 都会匹配失败

接下来的问题是，如果文本中不匹配的字符存在于待搜索串中要如何处理？为了不漏掉任何可能的解，我们只能向右少量移动，然后重新搜索，如图14.19所示。



(a) 待搜索串的最后一个字符‘e’和‘p’不匹配。但是‘p’出现在待搜索串中。



(b) 我们只能向右侧平移 2 个字符，然后重新检查。

图 14.19: 如果不匹配的字符出现在待搜索串中，需要向右侧少量平移

不匹配的字符很可能多次出现在待搜索串中。记待搜索串的长度为 $|P|$，该字符出现的位置依次为 $p_1, p_2, ..., p_i$。此时，我们需要用最后一个位置来计算平移的距离，以避免漏掉任何可能的解。

$$s = |P| - p_i \tag{14.50}$$

根据这一公式，待搜索串中的最后一个字符对应的平移距离为 0。在实现时，需要跳过这种情况。另外，由于平移的距离是根据待搜索串最后一个字符计算的（从 $|P|$ 减去相应的值），当从右向左扫描时，无论在哪里发生了不匹配，我们都要检查待搜索串中最后一个字符正对的文本中的字符，是否出现在不良字符表中。如图14.20所示。

在实际中，即使只使用不良字符规则也能够得到简单、快速的字符串查找算法，被称为 Boyer-Moore-Horspool 算法 [87]。

1: **procedure** BOYER-MOORE-HORSPOOL(T, P)

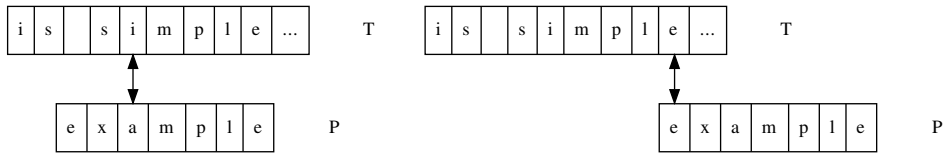


图 14.20: 即使字符‘i’和‘a’在中间位置匹配失败，我们要使用字符‘e’来查找平移的距离。得到结果 6（根据第一个‘e’出现的位置计算，需要跳过第二个‘e’出现的位置以避免平移距离为 0）

```
    for ∀c ∈ Σ do
        π[c] ← |P|
    for i ← 1 to |P| − 1 do                          ▷ 跳过最后一个位置
        π[P[i]] ← |P| − i
    s ← 0
    while s + |P| ≤ |T| do
        i ← |P|
        while i ≥ 1 ∧ P[i] = T[s + i] do             ▷ 从右侧开始扫描
            i ← i − 1
        if i < 1 then
            found one solution at s
            s ← s + 1                                ▷ 继续寻找下一个解
        else
            s ← s + π[T[s + |P|]]
```

记字符集为 Σ，我们首先将平移表的所有值都初始化为待搜索串的长度 $|P|$。然后，我们从左向右处理待搜索串，更新相应的平移距离。如果某个字符在待搜索串中多次出现，在右侧后出现的值将覆盖此前的值。开始查找时，我们将文本和待搜索串的左侧对齐。但是对于每个对齐的位置 s，我们都从右向左扫描，直到发生匹配失败，或者检查完待搜索串中的所有字符。后者说明我们发现了一个解；而对于前者，我们查找 π 并向右侧平移相应的距离。

下面的 Python 例子程序实现了这一算法。

```
def bmh_match(w, p):
    n = len(w)
    m = len(p)
    tab = [m for _ in range(256)] #保存不良字符规则的表
    for i in range(m-1):
        tab[ord(p[i])] = m - 1 - i
    res = []
    offset = 0
    while offset + m <= n:
```

```
        i = m - 1
        while i ≥ 0 and p[i] == w[offset+i]:
            i = i - 1
        if i < 0:
            res.append(offset)
            offset = offset + 1
        else:
            offset = offset + tab[ord(w[offset + m - 1])]
    return res
```

算法首先使用 $O(|\Sigma| + |P|)$ 的时间构造平移表格。如果字符集很小，则性能主要由待搜索串的长度和文本的长度决定。显然，最坏的情况下，文本和待搜索串中的所有字符都相同，例如在文本“aa.....a”（n 个字符‘a’，记为 a^n）中搜索“aa...a”（m 个字符‘a’，记为 a^m）。此时性能为 $O(mn)$。当待搜索的字符较长，并且有常数个解的时候，算法的性能良好，为线性时间。这一结论和后面介绍的完整 Boyer-Moore 算法在最好情况下的性能相同。

良好后缀启发条件

考虑在文本“bbbababbabab...”中搜索“abbabab”，如图14.21所示。根据不良字符规则，应该向右侧平移 2 个单位。

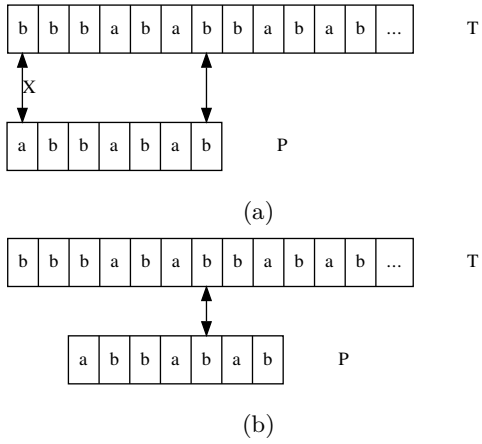


图 14.21: 根据不良字符规则，应向右平移 2 个单位，这样，下一个字符‘b’的位置相互对齐

实际上，我们可以做得更好。在匹配失败前，我们已经从右向左成功匹配了 6 个字符“bbabab”。由于“ab”既是待搜索串的前缀，也是已匹配部分的后缀，我们可以向右平移对齐这个后缀，如图14.22所示。

这和 KMP 算法中的预处理部分非常类似，但是我们不能总跳过这样多的字符。考虑如图14.23所示的例子。在失败前，我们已匹配了“bab”。虽然前缀“ab”也是“bab”的后缀，我们却不能平移这么远。这是因为“bab”也在其它位置出现过，即待搜索串的第 3 个字符的位置。为了避免漏掉可能的解，我们只能向右平移 2 个单位。

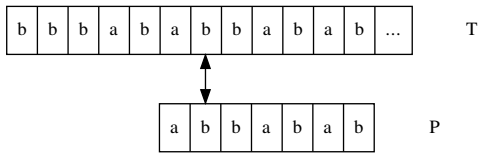


图 14.22: 由于前缀“ab”也是已匹配部分的后缀，我们可以向右平移使得“ab”对齐

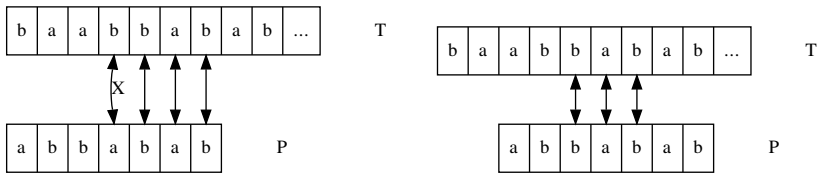


图 14.23: 已匹配的部分“bab”也在待搜索串的其它位置出现（从第 3 个字符到第 5 个字符）。我们只能向右平移 2 个单位以避免漏掉可能的解。

以上两种情况组成了良好后缀规则，如图14.24所示。

良好后缀规则用来处理多个字符已成功匹配的情况。如果下面任何一种情况发生，都可以向右平移一定的距离。

- 情况 1，如果已匹配部分的某个后缀同时也是待搜索字串的前缀，并且这一后缀不出现在待搜索字符串的其他位置，我们可以将待搜索串向右侧平移，对齐这一前缀；
- 情况 2，如果已匹配部分的某个后缀也出现在待搜索串的其它位置，我们可以将待搜索串向右侧平移，使得最右侧出现的位置对齐。

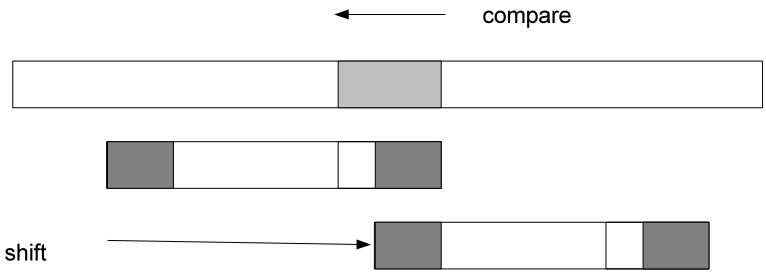
在扫描的过程中，只要可能，要优先使用第 2 种情况，如果发现已匹配的后缀没有出现过，然后再检查情况 1。由于良好后缀规则的两种情况都仅仅依赖于待搜索字符串，我们可以在搜索前进行预处理，构造出用于后继查询的表格。

简单起见,记 P 中从第 i 个字符开始的后缀为 $\overline{P_i}$。即 $\overline{P_i}$ 为子串 $P[i]P[i+1]...P[m]$。

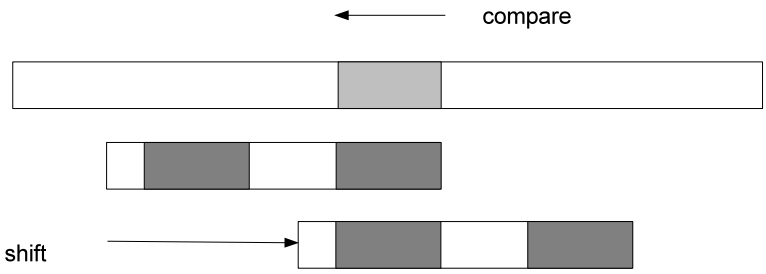
对于情况 1，我们可以检查 P 的每个后缀，包括 $\overline{P_m}$, $\overline{P_{m-1}}$, $\overline{P_{m-2}}$, ..., $\overline{P_2}$，看它是否同时是 P 的前缀。可以通过从右向左进行一轮扫描实现。

对于情况 2，我们可以检查 P 的每个前缀，包括 P_1, P_2, ..., P_{m-1}，看它的最长后缀是否也是 P 的后缀。可以通过从左向右的另一轮扫描实现。

```
1: function GOOD-SUFFIX(P)
2:     m ← |P|
3:     π_s ← {0, 0, ..., 0}                    ▷ 初始化一个长度为 m 的表格
4:     l ← 0                                   ▷ 最后的后缀也同时是 P 的前缀
5:     for i ← m − 1 down-to 1 do              ▷ 第一轮循环处理情况 1
6:         if P_i ⊏ P then                     ▷ ⊏ 代表左侧是右侧的前缀
```



(a) 情况 1，已匹配的子串中，有一部分也同时是待搜索串的前缀。



(b) 情况 2，匹配部分的后缀，也出现在待搜索串的其它位置。

图 14.24: 文本中浅灰色的部分代表已匹配的子串；深灰色的部分表示待搜索串中相同的内容

```
            l ← i
        π_s[i] ← l
    for i ← 1 to m do                                  ▷ 第二轮循环处理情况 2
        s ← Suffix-Length(P_i)
        if s ≠ 0 ∧ P[i − s] ≠ P[m − s] then
            π_s[m − s] ← m − i
    return π_s
```

这一算法构造良好后缀规则表 π_s。它首先检查 P 的每个后缀，从最短的开始，到最长的结束。如果后缀 $\overline{P_i}$ 同时是 P 的前缀，就将此后缀记录下来，并将其用于表格中所有的项，直到我们发现另一个后缀 $\overline{P_j}$，$j < i$ 并且同时是 P 的前缀。

然后，这一算法逐一检查 P 的所有前缀，从最短的开始，到最长的结束。它调用函数 Suffix-Length(P_i)，来计算 P_i 中最长的一个同时是 P 前缀的后缀的长度。如果长度 s 不等于 0，说明存在一个子串，同时也是待搜索串的后缀。它表明发生了情况 2。算法修改表格 π_s 从右侧数的第 s 项的值。为了避免再次找到已匹配的后缀，我们需要检查 $P[i-s]$ 和 $P[m-s]$ 是否相等。

函数 Suffix-Length 的实现如下。

```
function Suffix-Length(P_i)
    m ← |P|
    j ← 0
    while P[m − j] = P[i − j] ∧ j < i do
        j ← j + 1
    return j
```

下面的 Python 例子程序实现了良好后缀规则。

```
def good_suffix(p):
    m = len(p)
    tab = [0 for _ in range(m)]
    last = 0
    # 第一遍循环，针对情况 1
    for i in range(m-1, 0, -1): # m-1, m-2, ..., 1
        if is_prefix(p, i):
            last = i
        tab[i - 1] = last
    # 第二遍循环，针对情况 2
    for i in range(m):
        slen = suffix_len(p, i)
        if slen != 0 and p[i - slen] != p[m - 1 - slen]:
            tab[m - 1 - slen] = m - 1 - i
    return tab

# 检查 p[i...m-1] 是否是 p 的前缀
def is_prefix(p, i):
    for j in range(len(p) - i):
```

```
        if p[j] ≠ p [i+j]:
            return False
    return True

#返回最长后缀 p[...i] 的长度，它同时也是 p 的后缀
def suffix_len(p, i):
    m = len(p)
    j = 0
    while p[m - 1 - j] == p[i - j] and j < i:
        j = j + 1
    return j
```

当匹配失败时，不良字符规则和良好后缀规则可能同时适用。Boyer-Moore 算法比较这两种规则的结果，并选择较大的平移值以获得更快的速度。不良字符规则的表格可以按照如下的实现构造。

1: **function** BAD-CHARACTER(P)
2: **for** $\forall c \in \Sigma$ **do**
3: $\pi_b[c] \leftarrow |P|$
4: **for** $i \leftarrow 1$ to $|P| - 1$ **do**
5: $\pi_b[P[i]] \leftarrow |P| - i$
6: **return** π_b

下面的 Python 例子程序实现了不良字符规则表的构造算法。

```
def bad_char(p):
    m = len(p)
    tab = [m for _ in range(256)]
    for i in range(m-1):
        tab[ord(p[i])] = m - 1 - i
    return tab
```

最终的 Boyer-Moore 算法首先从待搜索串构造出两个规则表，将待搜索串和文本的左侧对齐，对每个对齐位置，都进行从右向左的扫描。如果不匹配发生，就尝试使用两种规则，并选择较大的距离向右侧平移。

1: **function** BOYER-MOORE(T, P)
2: $n \leftarrow |T|, m \leftarrow |P|$
3: $\pi_b \leftarrow$ BAD-CHARACTER(P)
4: $\pi_s \leftarrow$ GOOD-SUFFIX(P)
5: $s \leftarrow 0$
6: **while** $s + m \leq n$ **do**
7: $i \leftarrow m$
8: **while** $i \geq 1 \wedge P[i] = T[s + i]$ **do**
9: $i \leftarrow i - 1$
10: **if** $i < 1$ **then**

```
11:            found one solution at s
12:            s ← s + 1                                  ▷ 继续寻找下一个解
13:        else
14:            s ← s + max(π_b[T[s + m]], π_s[i])
```

下面的 Python 例子程序，完整地实现了 Boyer-Moore 算法。

```
def bm_match(w, p):
    n = len(w)
    m = len(p)
    tab1 = bad_char(p)
    tab2 = good_suffix(p)
    res = []
    offset = 0
    while offset + m ≤ n:
        i = m - 1
        while i ≥ 0 and p[i] == w[offset + i]:
            i = i - 1
        if i < 0:
            res.append(offset)
            offset = offset + 1
        else:
            offset = offset + max(tab1[ord(w[offset + m - 1])], tab2[i])
    return res
```

最初发表的 Boyer-Moore 算法，在最坏的情况下，只有当待搜索串不出现在文本中时，性能才是 $O(n+m)$[86]。在 1977 年，Knuth、Morris 和 Pratt 证明了这一结论。但是，当待搜索串出现在文本中时，如前所述，Boyer-Moore 算法在最坏情况下的性能为 $O(nm)$。

我们在此略过 Boyer-Moore 算法的纯函数式实现，读者可以参考 Richard Birds 给出的纯函数式 Boyer-Moore 算法（[1] 中的第 16 章,）。

练习 14.2

- 证明 Boyer-Moore 众数算法的正确性。
- 对于任意列表，寻找其中出现最多的元素。是否存在分而治之的解法？是否存在分而治之的数据结构，例如 map 可供使用？
- 如何找到一个列表中出现次数超过 1/3 的元素？如何找到一个列表中出现次数超过 1/m 的元素？
- 如果空数组不算合法的子数组，如何解决子数组最大和问题？
- Bentley 在 [2] 中给出了一个分而治之的方法求子数组最大和。复杂度为 $O(n \log n)$。思路是将列表在中点分成两份。我们可以递归地找出前半部分的最大和，和后半部分的最大和；但是我们还需要找出跨越中点部分的最大和，方法是从中点开始向左右两侧扫描：

```
1: function MAX-SUM(A)
2:     if A = φ then
3:         return 0
4:     else if |A| = 1 then
5:         return MAX(0, A[1])
6:     else
7:         m ← ⌊|A|/2⌋
8:         a ← MAX-FROM(REVERSE(A[1...m]))
9:         b ← MAX-FROM(A[m + 1...|A|])
10:        c ← MAX-SUM(A[1...m])
11:        d ← MAX-SUM(A[m + 1...|A|)
12:        return MAX(a + b, c, d)

13: function MAX-FROM(A)
14:    sum ← 0, m ← 0
15:    for i ← 1 to |A| do
16:        sum ← sum + A[i]
17:        m ← MAX(m, sum)
18:    return m
```

易知，这一方法存在性能关系 $T(n) = 2T(n/2) + O(n)$。选择一门编程语言，实现这一算法。

- 任给一个 $m \times n$ 的二维矩阵，矩阵中元素为整数，寻找其中的一个子矩阵，使得各元素相加后的和最大。
- 给定 n 个非负整数，用以表示一个一维等高地图，每个高度条的宽度都为 1，计算降雨后这一地形的积水数量。图14.25给出了一个例子。例如，等高地图数据

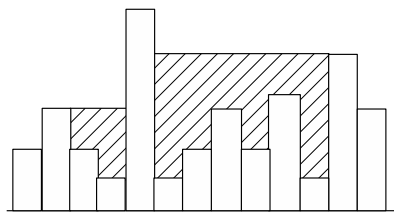


图 14.25: 灰色的区域表示积水

为 $\{0, 1, 0, 2, 1, 0, 1, 3, 2, 1, 2, 1\}$，则积水数量为 6。

- 解释在看起来“最坏”的情况下，为何 KMP 算法的性能仍然为线性?

- 使用逆序的 P_p 以避免线性时间的添加操作，改进实现纯函数式的 KMP 算法。
- 在文本 “anal” 中搜索字符串 “ananym”，试推导树 $left(right(right(right(T))))$ 的状态。

14.3 解的搜索

计算机程序可以用于解答某些趣题。在人工智能的早期阶段，人们发展出了搜索解的许多方法。和序列搜索、字符串匹配不同，问题的解并不一定直接存在于一个候选答案集中。往往需要一边构造解，一边进行尝试。某些问题可解，同时也存在大量无解的问题。即使是有解的问题，也通常存在多个解。例如，一个迷宫可能存在多种走出的路线。人们往往需要求出某种意义下的最优解。

14.3.1 深度优先搜索（DFS）和广度优先搜索（BFS）

DFS 和 BFS 分别代表深度优先搜索和广度优先搜索。它们通常作为图搜索算法加以介绍。图是一个很大的题目，超出了本书讲述的基本算法的范围。本节中，我们主要介绍如何使用 DFS 和 BFS 解决某些趣题，而不会正式介绍图的概念。

迷宫

迷宫的历史悠久，广受欢迎、是老少皆宜的一类趣题。图14.26给出了一个迷宫的例子。在某些公园，甚至还建有真正的迷宫供人游玩。在 1990 年代末，机器老鼠走迷宫的竞赛一度在世界上流行。

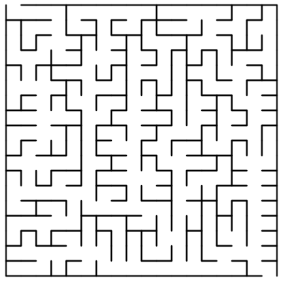


图 14.26: 一个迷宫的例子

迷宫有很多解法。本节将介绍一种有效的、但并非最好的方法。有很多针对迷宫解法的古老谚语，但是它们并非全都正确。

例如，有一个说法，当遇到分叉道路时，总向右转。如图14.27所示，这一招并不灵。明显可以先沿着上方的水平线前进，然后向右转，接着一直前进，经过 T 字路口就可到达终点。但如果遇到岔路就向右转，就会绕着中心的大方块不断转圈。

图 14.27: 如果一直右转，就会陷入循环

这个例子说明，当存在多个选项时，做出的决策会直接影响到最终的解。就像我们小时候读的童话故事，我们可以携带一块面包进入迷宫。当遇到岔路时，我们任选一条道路，然后留下一小块面包屑记录下这次尝试。如果我们遇到了死胡同，就沿着留下的面包屑向回走到上次做出选择的地方，然后换一条路。

任何时候，如果我们发现地上已经有面包屑了，就说明我们进入了循环，必须向回退然后重新尝试。不断重复这样的“尝试——检查”，我们或者最终找到走出迷宫的路，或者得知这个迷宫无解，此时我们最终回退到了迷宫的起点。

一种简单的描述迷宫的方法，是使用 $m \times n$ 的矩阵，每个元素的值为 0 或 1，表示这一位置是否有路。图14.27所示的迷宫可以用下面的矩阵定义。

$$
\begin{matrix}
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 1 & 1 & 1 & 0 \\
0 & 1 & 1 & 1 & 1 & 0 \\
0 & 1 & 1 & 1 & 1 & 0 \\
0 & 1 & 1 & 1 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 1 & 1 & 1 & 0
\end{matrix}
$$

给定起点 $s = (i, j)$ 和终点 $e = (p, q)$，我们要找出所有的解，也就是从 s 到 e 的全部路径。

显然存在一个递归的穷举搜索方法。为了找到所有从 s 到 e 的路径，我们可以检查和 s 连通的所有相邻点，对于每个点 k，我们递归找出从 k 到 e 的所有路径。这一方法可以描述如下。

- 边界情况，如果起点 s 和终点 e 相同，搜索结束；
- 否则，对所有和 s 连通的相邻点 k，递归找出从 k 到 e 的全部路径；如果可以通过 k 到达 e，将通路 s-k 连接到每个从 k 到 e 的路径前面。

但是，我们必须留下一些“面包屑”以避免重复尝试。否则，在递归的情况下，我们从 s 找到了一个连通点 k，然后我们继续寻找从 k 到 e 的路径。由于 s 同样和

k 连通，所以在接下来的递归中，我们将再次寻找从 s 到 e 的通路。这样就陷入了此前描述过的无穷循环中。

我们的解法是初始化一个空列表，用以记录我们走过的所有位置。对于每个连通的点，我们查找这一列表，看是否已经走过。我们跳过所有已走过的位置，而只尝试新的路径。对应的算法定义如下。

$$solveMaze(m, s, e) = solve(s, \{\phi\}) \tag{14.51}$$

其中 m 是定义迷宫的矩阵，s 是起点，e 是终点。函数 $solve$ 定义在 $solveMaze$ 的环境中，因此可以直接访问迷宫和终点。它的具体定义如下[8]。

$$solve(s, P) = \begin{cases} \{\{s\} \cup p | p \in P\} & : \quad s = e \\ concat(\{ \quad solve(s', \{\{s\} \cup p | p \in P\}) | & \\ \qquad s' \in adj(s), \neg visited(s')\}) & : \quad otherwise \end{cases} \tag{14.52}$$

这里 P 相当于一个累积器（accumulator）。每个连通的点都被记录在和当前位置连通的路径中。但是它们的顺序是逆序的，新走到的点被放在所有列表的头部，而起点被放在最后。这是因为列表的尾部添加操作是线性时间的（$O(n)$，其中 n 是列表中保存的元素个数），而在头部添加的操作是常数时间的。为了输出正常的路径顺序，我们可以将式 (14.51) 所有的解都反转[9]。

$$solveMaze(m, s, e) = map(reverse, solve(s, \{\phi\})) \tag{14.53}$$

接下来需要定义函数 $adj(p)$ 和 $visited(p)$，前者找出所有和点 p 相连通的点，后者检查点 p 是否以前已经尝试走过。如果矩阵中水平方向，或者垂直方向上的相邻元素，值都为 0，我们定义这两个点连通。

$$\begin{aligned} adj((x, y)) = \{(x', y') | \quad & (x', y') \in \{(x-1, y), (x+1, y), (x, y-1), (x, y+1)\}, \\ & 1 \le x' \le M, 1 \le y' \le N, m_{x'y'} = 0\} \end{aligned} \tag{14.54}$$

其中 M 和 N 分别是迷宫的宽和高。

函数 $visited(p)$ 检查点 p 是否已记录在列表 P 中的某一路径上。

$$visited(p) = \exists path \in P, p \in path \tag{14.55}$$

下面的 Haskell 例子程序实现了这一走迷宫算法。

```
solveMaze m from to = map reverse $ solve from [[]] where
    solve p paths | p == to = map (p:) paths
                  | otherwise = concat [solve p' (map (p:) paths) |
                                        p' ← adjacent p,
                                        not $ visited p' paths]
```

[8] 函数 $concat$ 可以将一组列表连接起来，例如：$concat(\{\{a, b, c\}, \{x, y, z\}\}) = \{a, b, c, x, y, z\}$。具体可以参见附录 A。

[9] $reverse$ 的具体定义可以参见附录 A。

```
    adjacent (x, y) = [(x', y') |
                       (x', y') ← [(x-1, y), (x+1, y), (x, y-1), (x, y+1)],
                       inRange (bounds m) (x', y'),
                       m ! (x', y') == 0]
    visited p paths = any (p `elem`) paths
```

对于下面由矩阵 mz 定义的迷宫，这一程序可以给出全部的解。

```
mz = [[0, 0, 1, 0, 1, 1],
      [1, 0, 1, 0, 1, 1],
      [1, 0, 0, 0, 0, 0],
      [1, 1, 0, 1, 1, 1],
      [0, 0, 0, 0, 0, 0],
      [0, 0, 0, 1, 1, 0]]

maze  = listArray ((1,1), (6, 6)) ∘ concat

solveMaze (maze mz) (1,1) (6, 6)
```

我们此前提到，这是一种“穷举搜索”的解法，它递归地搜索所有连通的点作为候选。在实际的迷宫竞赛中，例如机器老鼠走迷宫竞赛，找到一条路径就足够了。我们可以调整解法，它和本节开始时描述的方法类似，机器老鼠总是选择第一个连通点，而跳过其它选择直到无法前进。我们需要某种数据结构保存“面包屑”，记录此前做出的决策。由于我们总是在最新的决策基础上搜索通路，因此是后进先出的顺序。我们可以使用一个栈来实现。

在开始的时候，只有起点 s 保存在栈中。我们将其弹出，找出和 s 相连通的点，例如 a 和 b。然后我们将两条可能的路径 $\{a, s\}$ 和 $\{b, s\}$ 推入栈中。接下来，我们将路径 $\{a, s\}$ 弹出，然后检查和点 a 相连通的点。然后所有经过 3 步可到达的路径被推回栈。我们重复这一过程。任何时候，栈中的每个元素都代表一条逆序的路径，它从起点开始，通向可到达的最远位置。如图14.28所示。

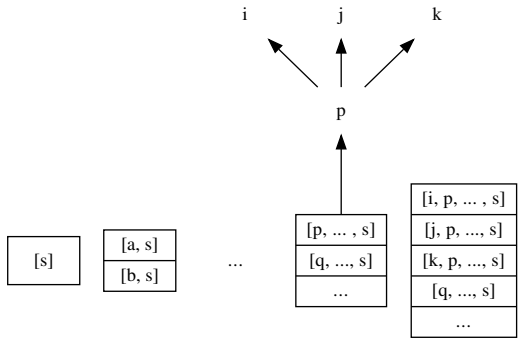


图 14.28: 栈初始时包含一个只有一个元素的列表。这一元素为起点 s。s 和点 a、b 连通。路径 $\{a, s\}$ 和 $\{b, s\}$ 被推回栈。在某一步，以 p 结束的路径被弹出。p 和点 i、j 和 k 连通。这 3 个点被扩展为不同的选项，并推回到栈中。除非所有的候选路径都失败，否则不会尝试以 q 结尾的候选路径。

栈可以用一个列表来实现。最新的选项可以从表头获得，新的候选路径也被添加到表头。可以通过这样的路径列表解决迷宫问题。

$$solveMaze'(m, s, e) = reverse(solve'(\{\{s\}\})) \tag{14.56}$$

由于我们搜索第一个，而不是全部的解，这里我们没有使用 map 函数。当栈为空时，表示我们已经尝试了所有的可能，但仍然没有找到通路。因此迷宫无解；否则，我们弹出栈顶的候选路径，将其扩展到所有未曾走过的连通点，然后再推回栈。我们用 S 表示栈，若栈不为空，则栈顶的元素记为 s_1，弹出栈顶元素后的新栈表示为 S'。s_1 为一个点的列表，代表路径 P。记这条路径中的第一个点为 p_1，其余的点为 P'。这一解法可以定义如下。

$$solve'(S) = \begin{cases} \phi & : & S = \phi \\ s_1 & : & s_1 = e \\ solve'(S') & : & C = \{c | c \in adj(p_1), c \notin P'\} = \phi \\ solve'(\{\{p\} \cup P | p \in C\} \cup S) & : & C \neq \phi \end{cases} \tag{14.57}$$

其中 adj 的定义和前面相同。下面的 Haskell 例子程序实现了这一迷宫算法[10]。

```
dfsSolve m from to = reverse $ solve [[from]] where
    solve [] = []
    solve (c@(p:path):cs)
        | p == to = c -- 找到第一个解后结束
        | otherwise = let os = filter (`notElem` path) (adjacent p) in
                      if os == []
                      then solve cs
                      else solve ((map (:c) os) ++ cs)
```

可以很容易地修改这一算法，从而找到全部的解。在第二行找到一个解后，我们不立即返回，而是将其记录下来，然后继续尝试栈中记录的其他候选路径，直到栈变为空。我们将其作为练习留给读者。

也可以用命令式的方法实现这一思路。我们使用一个栈保存从起点开始的全部可能路径。每次迭代，首先弹出栈顶保存的路径，如果这一路径到达了终点，则找到了迷宫的一个解；否则，我们将尚未尝试过的所有连通点添加到路径上作为新的候选路径，并推回栈。重复这一过程直到栈中的所有候选路径都检查完毕。

我们使用同样的符号 S 表示栈。但在命令式的环境中，路径使用数组来表示，这样效率更高。为此，起点保存在数组的第一个元素中，而最远到达的点保存为最右侧的元素。我们用 P_n 来表示路径 P 中的最后一个元素 $\text{LAST}(P)$。命令式的算法定义如下。

1: **function** SOLVE-MAZE(m, s, e)
2: $S \leftarrow \phi$
3: PUSH($S, \{s\}$)

[10] $adjacent$ 函数的定义完全相同，在此略过。

```
    L ← φ                                                    ▷ 结果列表
    while S ≠ φ do
        P ← POP(S)
        if e = p_n then
            ADD(L, P)
        else
            for ∀p ∈ ADJACENT(m, p_n) do
                if p ∉ P then
                    PUSH(S, P ∪ {p})
    return L
```

下面的 Python 例子程序实现了这一迷宫算法。

```
def solve(m, src, dst):
    stack = [[src]]
    s = []
    while stack ≠ []:
        path = stack.pop()
        if path[-1] == dst:
            s.append(path)
        else:
            for p in adjacent(m, path[-1]):
                if not p in path:
                    stack.append(path + [p])
    return s

def adjacent(m, p):
    (x, y) = p
    ds = [(0, 1), (0, -1), (1, 0), (-1, 0)]
    ps = []
    for (dx, dy) in ds:
        x1 = x + dx
        y1 = y + dy
        if 0 ≤ x1 and x1 < len(m[0]) and
           0 ≤ y1 and y1 < len(m) and m[y][x] == 0:
            ps.append((x1, y1))
    return ps
```

同样的例子迷宫可以用这一程序解决如下。

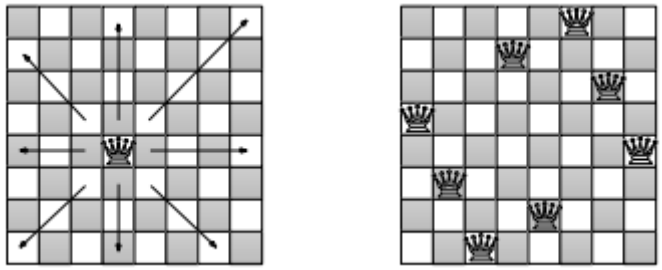
```
mz = [[0, 0, 1, 0, 1, 1],
      [1, 0, 1, 0, 1, 1],
      [1, 0, 0, 0, 0, 0],
      [1, 1, 0, 1, 1, 1],
      [0, 0, 0, 0, 0, 0],
      [0, 0, 0, 1, 1, 0]]

solve(mz, (0, 0), (5,5))
```

看上去在最坏的情况下，每步都有上下左右 4 个选项，每个选项都被推入栈，并

且最终在回溯时都被检查了。算法的复杂度看似是 $O(4^n)$。实际上消耗的时间并不会这样大，这是因为我们过滤掉了已经走过的位置。在最坏情况下，所有可以到达的点都恰好被访问过一次。因此时间复杂度为 $O(n)$，其中 n 是互相连通的点的数量。由于使用了一个栈来保存候选路径，空间复杂度为 $O(n^2)$。

八皇后问题

八皇后问题是一个很著名的趣题。虽然国际象棋有着悠久的历史，但八皇后趣题直到 1848 年才由 Max Bezzel 提出 [89]。皇后是国际象棋中一种威力巨大的棋子。她可以攻击在同一行、列和斜线上的任意距离的其它棋子。这道趣题要求找到一种方法，可以在棋盘上同时摆下八个皇后，而她们之间不会互相攻击。图14.29 (a) 描述了皇后可以攻击到的范围。图14.29 (b) 给出了八皇后问题的某一种解。



(a) 国际象棋中的皇后。　　(b) 某一种解。

图 14.29: 八皇后问题

显然，可以用暴力方法穷举解决八皇后问题，在国际象棋棋盘的 64 个格子中，放入 8 个皇后，这需要在 P_{64}^8 个可能的排列中检查。这个数字大约为 4×10^{10}。显然我们可以改进这一方法，考虑任一行中不能包含 2 个及以上的皇后，并且任何一个皇后都必须放在第 1 列到第 8 列中的某一列上，所以一个解的布局必然是 $\{1,2,3,4,5,6,7,8\}$ 的某种排列。例如布局 $\{6,2,7,1,3,5,8,4\}$ 表示，第一个皇后摆放在第 1 行、第 6 列上；第二个皇后摆在第 2 行、第 2 列上……最后一个皇后摆在第 8 行、第 4 列上。通过这一方法，我们只需要检查 $8! = 40320$ 种可能的布局。

我们可以找到更好的解法。和迷宫问题类似，我们可以从第一行开始，逐一摆放皇后。对于第一个皇后，存在 8 种可能的摆法，她可以被放置在八列中的某一列上。接下来摆放第二个皇后，我们检查 8 个可能的列。由于可能被第一个皇后攻击，因此某些列不能再摆放了。我们重复这一过程，对于第 i 个皇后，我们检查第 i 行中的 8 个位置，找到不被任何前 $i-1$ 个皇后攻击的位置。如果所有 8 个位置都不能摆放，即这一行的 8 个位置都会被此前摆放过的某个皇后攻击，我们就必须向迷宫问题中一样进行回溯。当所有 8 个皇后都成功放入棋盘后，我们就找到了一个可行的解。为了找到所有可能解，我们需要记录下这一布局，然后继续检查其他可能的列，并进行必要的回溯。当第一行的 8 列都尝试完毕后，这一过程结束。下面的函数启动八皇后

问题解的查找过程。

$$solve(\{\phi\}, \phi) \tag{14.58}$$

和迷宫问题类似，我们使用一个栈 S 来记录可能的尝试。一开始栈中只有一个空元素。我们使用一个列表 L 来记录所有可行的解。记栈顶的元素为 s_1，它是某种尚未完成的布局，也就是 1 到 8 中部分元素的排列。将栈顶元素 s_1 弹出后，剩下的部分记为 S'。函数 $solve$ 的具体定义如下。

$$solve(S, L) = \begin{cases} L & : \quad S = \phi \\ solve(S', \{s_1\} \cup L) & : \quad |s_1| = 8 \\ solve(\begin{Bmatrix} \{i\} \cup s_1 | & i \in [1, 8], \\ & i \notin s_1, \\ & safe(i, s_1) \end{Bmatrix} \cup S', L) & : \quad otherwise \end{cases} \tag{14.59}$$

若栈为空，表明所有可能都已经尝试完毕，我们已无法继续回溯了。列表 L 已记录下了所有找到的解，我们将其作为结果返回；否则，若栈顶元素所代表的布局长度为 8，表明我们找到了一种可行的解。我们将其记录到 L 中，然后继续寻找其它的解；如果这一布局的长度小于 8，表明我们需要继续摆放剩余的皇后。我们从第 1 到第 8 列中，找出尚未被占的列（通过 $i \notin s_1$ 条件），同时它不能被斜线上的其他皇后攻击（通过 $safe$ 条件）。可行的布局被推入栈中用于此后的搜索。

函数 $safe(x, C)$ 检查在位置 x 上的皇后是否会被 C 中的任意皇后从斜线方向攻击。有两种可能的情况，分别是 45° 度和 135° 度方向。由于这一皇后所在的行为 $y = 1 + |C|$，其中 $|C|$ 是中间布局 C 的长度，因此函数 $safe$ 可定义如下。

$$safe(x, C) = \forall (c, r) \in zip(reverse(C), \{1, 2, ...\}), |x - c| \neq |y - r| \tag{14.60}$$

其中 zip 将两个列表中的每个元素都结合成一对，组成一个新的列表。因此，若 $C = \{c_{i-1}, c_{i-2}, ..., c_2, c_1\}$ 代表前 $i - 1$ 个皇后分别所在的列，上述函数将检查每个皇后的行列位置 $\{(c_1, 1), (c_2, 2), ..., (c_{i-1}, i - 1)\}$ 是否会和位置 (x, y) 构成对角线。

下面的 Haskell 例子程序实现了这一八皇后问题的解。

```
solve = dfsSolve [[]] [] where
    dfsSolve [] s = s
    dfsSolve (c:cs) s
             | length c == 8 = dfsSolve cs (c:s)
             | otherwise = dfsSolve ([(x:c) | x ← [1..8] \\ c,
                                not $ attack x c] ++ cs) s
    attack x cs = let y = 1 + length cs in
                  any (λ(c, r) → abs(x - c) == abs(y - r)) $
                      zip (reverse cs) [1..]
```

观察到这一算法是尾递归的，它可以很容易地用命令式的方式实现。我们使用数组而非列表来表示皇后的布局。记栈为 S，中间布局为 A，命令式算法可以描述如下。

```
1: function SOLVE-QUEENS
2:     S ← {φ}
3:     L ← φ                                          ▷ 保存所有解的列表
4:     while S ≠ φ do
5:         A ← POP(S)                                 ▷ A 是某一中间布局
6:         if |A| = 8 then
7:             ADD(L, A)
8:         else
9:             for i ← 1 to 8 do
10:                if VALID(i, A) then
11:                    PUSH(S, A ∪ {i})
12:    return L
```

栈中一开始放入一个空布局。然后不断取出栈顶元素，如果还有皇后尚未摆放完毕，我们就依次检查下一行中的所有 8 个位置。如果该位置是安全的，也就是说它不被此前的任意皇后攻击，就将此位置添加到布局中，并推回栈。和函数式方法不同，由于使用数组，我们无需再将解的布局反转。

函数 VALID 检查中间布局 A 中的下一行的 x 列位置摆放皇后是否安全。它去掉已经被占的列，然后计算对角线上是否有别的皇后。

```
1: function VALID(x, A)
2:     y ← 1 + |A|
3:     for i ← 1 to |A| do
4:         if x = A[i] ∨ |y − i| = |x − A[i]| then
5:             return False
6:     return True
```

下面的 Python 例子程序实现了这一命令式八皇后解法。

```
def solve():
    stack = [[]]
    s = []
    while stack ≠ []:
        a = stack.pop()
        if len(a) == 8:
            s.append(a)
        else:
            for i in range(1, 9):
                if valid(i, a):
                    stack.append(a+[i])
    return s

def valid(x, a):
    y = len(a) + 1
    for i in range(1, y):
        if x == a[i-1] or abs(y - i) == abs(x - a[i-1]):
```

```
            return False
    return True
```

虽然摆放每个皇后时有 8 列可供选择，但是并非所有列都可行。只有此前没有被占的列才会被尝试。算法只检查 15720 种情况，这要远远小于 $8^8 = 16777216$ 种可能[89]。

可以很容易将这一算法加以扩展，用以解决 n 皇后问题，其中 $n \geq 4$。但是随着 n 的增大，所用的时间急速增加。这一回溯算法仅仅比枚举 1 到 8 的全排列稍快（枚举全排列的时间是 $o(n!)$）。此外，还存在另一种小改进，由于国际象棋棋盘是正方形的，它水平方向和垂直方向都对称。因此得到一个解后，通过旋转和翻转，可以得到其他对称的解。我们将这一改进留给读者作为练习。

跳棋趣题

我曾经收到过一道关于青蛙的趣题。据说这是中国二年级小学生的家庭作业。如图14.30所示，在 7 块排成一排的石头上有 6 只青蛙。如果前方的石头是空的，青蛙可以跳到石头上；青蛙还可以越过一只青蛙，跳到前方的空石头上。左侧的青蛙只能向右侧前进，而右侧的青蛙只能向左侧前进。图14.31描述了青蛙跳跃的规则。

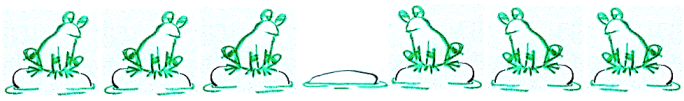


图 14.30: 跳跃的青蛙趣题

这道题目要求按照规则安排青蛙移动或者跳跃，使得左右的 3 只青蛙位置互换。如果我们标记左侧的青蛙为 A，右侧的为 B，没有青蛙的石头为 O，这道题目就是要求找到解使得从 AAAOBBB 转换到 BBBOAAA。

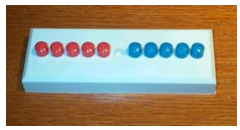
(a) 跳到相邻的石头上。

(b) 向右侧越过一只青蛙。

(c) 向左侧越过一只青蛙。

图 14.31: 移动规则

这道趣题是跳棋类趣题的一种特殊形式。跳棋的个数并不一定限制为 6，它可以是 8 或者更大的偶数。图14.32给出了一些这类问题的变化形式。



(a) 单人跳棋（Solitaire）

(b) Hop over

(c) Draught board

图 14.32: 跳棋趣题的变化形式，来自 http://www.robspuzzlepage.com/jumping.htm

我们可以通过编程的方法解决这类趣题。思路和八皇后问题类似。记从左向右的石头位置为 1, 2, ..., 7。理想情况下，有 4 种可能的移动。例如游戏开始的时候，第 3 块石头上的青蛙可以移动到空石头上；对称地，第 5 块石头上的青蛙也可以向左移动一步；另外，第 2 块石头上的青蛙可以向右越过一只青蛙，跳到空石头上，同样，第 6 块石头上的青蛙，也可以向左越过一只青蛙。

每走一步，我们可以记录下青蛙们的状态，然后尝试 4 种方案中的一种。当然并非任何时候，4 种方案都可行。如果我们走不下去了，就回溯并尝试其它方案。

由于我们限制左侧的青蛙只能向右，右侧的青蛙只能向左，因此这些移动都是不可逆的。和迷宫游戏不同，这里不可能存在重复的情况。但是，我们仍需记录移动的步骤，以便最后的输出。

为了强调这些条件，我们分别用-1、0、和 1 代表 A、O、和 B。一个状态就是一列元素，每个元素是这 3 个值中的一种。起始状态为 $\{-1,-1,-1,0,1,1,1\}$。$L[i]$ 表示第 i 个元素，它的值表明第 i 个石头是否为空，或者存在一只左侧移动来的青蛙，或者存在一只右侧移动来的青蛙。记空石头的位置为 p。4 种可能的移动方案可以描述如下。

- 向左跳跃（Leap left）：$p < 6$，且 $L[p+2] > 0$，交换 $L[p] \leftrightarrow L[p+2]$；
- 向左移动（Hop left）：$p < 7$，且 $L[p+1] > 0$，交换 $L[p] \leftrightarrow L[p+1]$；
- 向右跳跃（Leap right）：$p > 2$，且 $L[p-2] < 0$，交换 $L[p-2] \leftrightarrow L[p]$；
- 向右移动（Hop right）：$p > 1$，且 $L[p-1] < 0$，交换 $L[p-1] \leftrightarrow L[p]$。

为此，我们定义 4 个函数 $leap_l(L)$、$hop_l(L)$、$leap_r(L)$、和 $hop_r(L)$。若 L 不满足移动的条件，这些函数将返回同样的 L，否则，它们返回变化后的状态 L'。

我们可以使用一个栈 S 来保存已做过的尝试。开始的时候，栈中包含一个列表，列表中只有一个元素，就是开始状态。我们将找到的解保存在列表 M 中，M 起始为空。

$$solve(\{\{-1,-1,-1,0,1,1,1\}\},\phi) \tag{14.61}$$

只要栈不为空,我们就取出栈顶元素。如果最后的状态等于 $\{1,1,1,0,-1,-1,-1\}$，说明找到了一个解。我们将直到这一状态的一系列移动方案添加到 M 中；否则，我

们在最后的状态上尝试 4 种可能的移动，并将可行的移动方法推回栈以便将来继续搜索。记堆栈为 S，栈顶的元素为为 s_1，s_1 中记录的最后的状态为 L。算法可以定义如下。

$$solve(S, M) = \begin{cases} M & : & S = \phi \\ solve(S', \{reverse(s_1)\} \cup M) & : & L = \{1, 1, 1, 0, -1, -1, -1\} \\ solve(P \cup S', M) & : & otherwise \end{cases} \tag{14.62}$$

其中 P 是在最后的状态 L 之上可能的移动方法:

$$P = \{L' | L' \in \{leap_l(L), hop_l(L), leap_r(L), hop_r(L)\}, L \neq L'\} \tag{14.63}$$

起始状态被保存为最后一个元素，而最后的状态是第一个元素。因此我们需要将其反转，保存在解的列表中。

下面的 Haskell 例子程序，实现了跳跃青蛙问题的解。

```
solve = dfsSolve [[[-1, -1, -1, 0, 1, 1, 1]]] [] where
    dfsSolve [] s = s
    dfsSolve (c:cs) s
             | head c == [1, 1, 1, 0, -1, -1, -1] = dfsSolve cs (reverse c:s)
             | otherwise = dfsSolve ((map (:c) $ moves $ head c) ++ cs) s

moves s = filter (≠s) [leapLeft s, hopLeft s, leapRight s, hopRight s] where
    leapLeft [] = []
    leapLeft (0:y:1:ys) = 1:y:0:ys
    leapLeft (y:ys) = y:leapLeft ys
    hopLeft [] = []
    hopLeft (0:1:ys) = 1:0:ys
    hopLeft (y:ys) = y:hopLeft ys
    leapRight [] = []
    leapRight (-1:y:0:ys) = 0:y:(-1):ys
    leapRight (y:ys) = y:leapRight ys
    hopRight [] = []
    hopRight (-1:0:ys) = 0:(-1):ys
    hopRight (y:ys) = y:hopRight ys
```

运行这一程序可以找出 2 个对称的解，每个都需要 15 步。下表列出了其中的一个解。

观察上述算法，它是尾递归的，因此可以较容易地用命令式方式实现。我们将算法扩展为解决每侧有 n 只青蛙的题目。记起始状态 s 为 {-1, -1, ..., -1, 0, 1, 1, ..., 1}，左右翻转后的终止状态为 e。

1: **function** SOLVE(s, e)
2: $S \leftarrow \{\{s\}\}$
3: $M \leftarrow \phi$
4: **while** $S \neq \phi$ **do**
5: $s_1 \leftarrow$ POP(S)

step	-1	-1	-1	0	1	1	1
1	-1	-1	0	-1	1	1	1
2	-1	-1	1	-1	0	1	1
3	-1	-1	1	-1	1	0	1
4	-1	-1	1	0	1	-1	1
5	-1	0	1	-1	1	-1	1
6	0	-1	1	-1	1	-1	1
7	1	-1	0	-1	1	-1	1
8	1	-1	1	-1	0	-1	1
9	1	-1	1	-1	1	-1	0
10	1	-1	1	-1	1	0	-1
11	1	-1	1	0	1	-1	-1
12	1	0	1	-1	1	-1	-1
13	1	1	0	-1	1	-1	-1
14	1	1	1	-1	0	-1	-1
15	1	1	1	0	-1	-1	-1

表 14.6: 青蛙趣题的一个解

6: **if** $s_1[1] = e$ **then**
7: ADD(M, REVERSE(s_1))
8: **else**
9: **for** $\forall m \in$ MOVES($s_1[1]$) **do**
10: PUSH(S, $\{m\} \cup s_1$)
11: **return** M

可能的移动方法可以被实现为 MOVES 过程。它可以处理任意只青蛙的情况。下面的 Python 程序实现了这一解法。

```
def solve(start, end):
    stack = [[start]]
    s = []
    while stack ≠ []:
        c = stack.pop()
        if c[0] == end:
            s.append(reversed(c))
        else:
            for m in moves(c[0]):
                stack.append([m]+c)
    return s

def moves(s):
    ms = []
    n = len(s)
```

```
    p = s.index(0)
    if p < n - 2 and s[p+2] > 0:
        ms.append(swap(s, p, p+2))
    if p < n - 1 and s[p+1] > 0:
        ms.append(swap(s, p, p+1))
    if p > 1 and s[p-2] < 0:
        ms.append(swap(s, p, p-2))
    if p > 0 and s[p-1] < 0:
        ms.append(swap(s, p, p-1))
    return ms

def swap(s, i, j):
    a = s[:]
    (a[i], a[j]) = (a[j], a[i])
    return a
```

对于每侧有 3 只青蛙的情况，我们知道共需要 15 步才能让它们左右互换。通过上述算法，我们可以得到解法的步数和每侧青蛙数目的一个关系，如下表：

每侧青蛙的数目	1	2	3	4	5	...
解法的步数	3	8	15	24	35	...

表 14.7: 青蛙数目和解法步数的对应关系表

表中列出的解法的步数恰好是完全平方数减一。因此我们猜测，解法的步数和每侧青蛙的数目 n 的关系为 $(n+1)^2-1$。实际上，我们可以证明这一点。

比较最终的状态和最初的状态，每只青蛙都向相对的一侧移动了 $n+1$ 块石头。因此 $2n$ 只青蛙，总共移动了 $2n(n+1)$ 块石头。另一个重要的事实是，左侧的每只青蛙，必然和右侧的所有青蛙相遇一次。一旦相遇，必然发生一次跳跃。由于一共有 n^2 次相遇，因此共导致了所有青蛙前进了 $2n^2$ 块石头。剩下的移动不是跳跃，而是跳到相邻的石头上，总共有 $2n(n+1)-2n^2=2n$ 次。将 n^2 次跳跃，和 $2n$ 次跳到相邻石头上相加。得到最终解的步数为：$n^2+2n=(n+1)^2-1$。

深度优先搜索的小结

观察上述 3 个趣题，虽然它们各不相同，但是它们的解法却有着类似的结构。它们都有着某种起始状态。迷宫问题从入口开始；八皇后问题从空棋盘开始；跳跃青蛙问题从 AAAOBBB 的状态开始。解的过程是一种搜索，每次尝试，都有若干种可能的选项。迷宫问题中，每走一步都有上下左右四个方向可供选择；八皇后问题中，每次摆放都有 8 列可供选择；跳跃青蛙趣题中，每次尝试都有 4 种不同的跳跃方式可供选择。虽然每次选择，我们都不知能继续走多远。但我们始终清楚地知道最终状态是什么。在迷宫问题中，最终状态是出口；八皇后问题中，最终状态是 8 个皇后都摆放在棋盘上；跳跃青蛙趣题中，最终状态是所有青蛙的位置互换。

我们使用相同的策略来解决这些问题。我们不断选择可能的选项尝试，记录已经

达到的状态，如果无法继续就进行回溯并尝试其它选项。通过这样的方法，我们或者可以找到解，或者穷尽所有可能而发现问题无解。

当然，这类解法还存在一些变化，当找到一个解后，我们可以停下结束，或者继续寻找所有可能的解。

如果我们以起始状态为根，画出一棵树，每个树枝代表一个不同的选择。我们的搜索过程，是一个不断深入的过程。只要能够继续，我们就不考虑同一深度上的其它选项。直到失败后回溯到树的上一层。图14.33描述了我们在状态树中的搜索顺序。箭头方向表明了我们如何先向下，在向上回溯的过程。节点上的数字是我们访问它们的顺序。

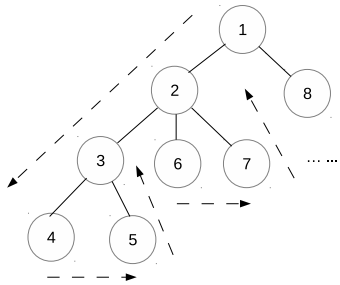


图 14.33: 深度优先搜索的顺序

这样的搜索策略称为深度优先搜索 DFS（Deep-first-search)。在现实世界中，我们在不经意间广泛使用深度优先搜索。某些编程环境，例如 Prolog，使用深度优先作为默认的求值模型。例如一个迷宫可以被一组规则描述：

```
c(a, b). c(a, e).
c(b, c). c(b, f).
c(e, d), c(e, f).
c(f, c).
c(g, d). c(g, h).
c(h, f).
```

其中，断言 $c(X, Y)$ 表示位置 X 和 Y 连通。注意，这一断言是有方向性的。如果要让 Y 和 X 连通，我们可以增加一条对称的断言，或者建立一条无方向性的断言。图14.34给出了一个有向图。任意给出两个位置 X 和 Y，Prolog 可以通过下面的程序判定它们之间是否有通路。

```
go(X, X).
go(X, Y) :- c(X, Z), go(Z, Y)
```

这一程序说明，一个位置和自己相通。任意给出两个不同的位置 X 和 Y，若 X 和 Z 相连，且 Z 和 Y 之间有通路，则 X 和 Y 之间存在通路。显然，Z 的选择可

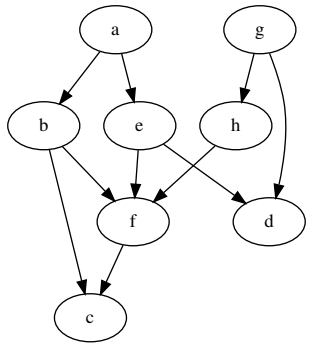


图 14.34: 一个有向图

能是不唯一的。Prolog 会选择一个，然后继续进行搜索。只有当递归搜索失败时，才会尝试其它选择。此时，Prolog 会回溯，并更换到下一个选项上。这恰好就是深度优先的搜索策略。

当我们只需要找到解，而并不关心找到最少步数的解时，深度优先搜索是很有效的方法。例如，迷宫问题中找出的第一个解并不一定是最短的路径。我们接下来将讨论更多的趣题，并给出找出最少步数解的方法。

狼、羊、白菜趣题

这是一道传统趣题。有一个农夫，带着一只狼、一只羊、和一筐白菜要过河。有一条小船，只有农夫会划船。由于船很小，只能装下农夫和另外一样东西。农夫每次只能在狼、羊、白菜中任选一样和他一起过河。但是如果农夫不在，狼会吃掉羊，而羊会吃掉白菜。这道题目要求找到最快的一种方法，可以让所有的东西都渡过河。

图 14.35: 狼、羊、白菜问题

这道题目的关键是狼不会吃掉白菜。因此农夫可以安全地将羊运到河对岸，并返

回。但是接下来，无论他将狼或白菜中的任何一样运过河，他必须将某一样运回以避免有东西被吃掉。为了寻找最快的解法，只要存在多种选择，我们可以并发检查所有的选项，比较哪个会更快。如果不考虑渡河的方向，只要渡过一次，就算做一步，往返算两步，我们实际上在检查渡河一次后的所有可能、渡河两次后所有可能、三次后的所有可能……直到某次后，我们发现所有的东西都到达了河对岸，这一过程结束。并且这一渡河方法在所有可能中胜出，是最快的解法。

问题在于，我们无法真正并发检查所有可能的解法。除非使用带有多个 CPU 内核的超级计算机，但是对于解决这样一道简单的趣题，这相当于“高射炮打蚊子”。

让我们考虑一个抽奖游戏。游戏参与者不能看，闭着眼睛从一个箱子里掏出一个球。箱子里只有一个黑色球，其余的球都是白色的。摸到黑球的人获胜；如果摸到白球，他必须把球放回箱子，然后等待下次摸球。为了使得游戏公平，我们可以指定这样一个规则：必须等待所有其他人都摸过之后，才能再摸第二次。我们可以让参与游戏的人站成一队。每次站在队伍前面的人摸球，如果他没有摸到黑球获胜，他就站到队尾等待下次摸球。这一队列可以保证游戏的公平规则。

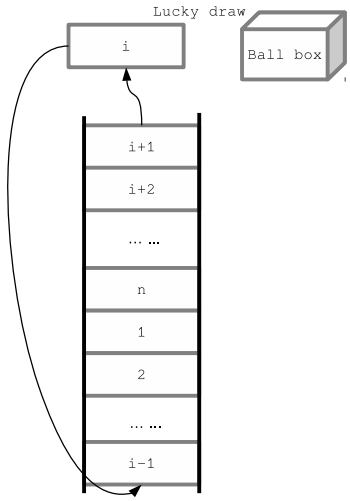


图 14.36: 抽奖游戏，第 i 个人出队，摸球。如果没有摸到黑球，就站到队尾

我们可以用类似的思路来解决狼、羊、白菜趣题。河的两岸可以用两个集合 A 和 B 代表。开始的时候，集合 A 中包含狼、羊、白菜、和农夫；而集合 B 是空集。我们每次将农夫和另外一个元素从一个集合移动到另一个集合。每个集合中，如果不存在农夫，则不能含有相互冲突的东西。目标是用最少的次数，交换 A 和 B 的内容。

我们使用一个队列，最开始只包含一个状态 $A = \{w, g, c, p\}$、$B = \phi$。只要队列不为空，我们就取出队列头部的元素，将其扩展为所有可能的选择，然后将扩展后的候选状态放回队列尾部。如果队列头部的第一个元素就是最终的目标，即 $A = \phi$、

$B = \{w, g, c, p\}$，我们就找到了解。图14.37描述了这一思路的搜索顺序。同一深度上的所有可能性都被检查了，因此无需进行回溯。

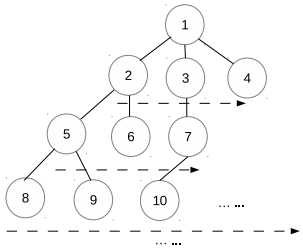


图 14.37: 从第一个状态开始，检查第二步的所有选项 2、3、和 4；然后检查第 3 层上的所有选项……

我们可以用一个 4 位二进制数来表示集合，每一位表示一种事物，例如狼 $w = 1$、羊 $g = 2$、白菜 $c = 4$、农夫 $p = 8$。0 表示空集合，15 表示包含所有事物的集合。值 3 表示只有狼和羊被留在了河的这一侧。此时狼会吃掉羊。同样，值 6 表示另外一种存在冲突的情况。每次我们将最高位（值为 8）和另外一位（4、2、或 1）从一个数字移动到另外一个数字上。可行的移动方法定义如下：

$$mv(A, B) = \begin{cases} \{(A - 8 - i, B + 8 + i) | i \in \{0, 1, 2, 4\}, i = 0 \vee A \barwedge i \neq 0\} & : \quad B < 8 \\ \{(A + 8 + i, B - 8 - i) | i \in \{0, 1, 2, 4\}, i = 0 \vee B \barwedge i \neq 0\} & : \quad Otherwise \end{cases} \tag{14.64}$$

其中 $\barwedge$ 表示按位与运算。

我们可以使用前面章节定义的纯函数式队列。记队列为 Q，最开始队列包含一个列表，列表只含有一对元素 $\{(15, 0)\}$。若 Q 不为空，则函数 $DeQ(Q)$ 取出队列的头部元素 M，队列中的剩余元素记为 Q'。M 为包含若干对元素的列表，代表在河岸间的一系列移动。表中第一个元素为 $m_1 = (A', B')$，是最后一次移动后的状态。函数 $EnQ'(Q, L)$，是一个稍作改动的入队操作。它将 L 中所有可能的移动序列，逐一加入到队列的尾部，并返回新的队列。使用这些记号，寻找解的算法可以定义为如下的函数。

$$solve(Q) = \begin{cases} \phi & : \quad Q = \phi \\ reverse(M) & : \quad A' = 0 \\ solve(EnQ'(Q', \left\{ \{m\} \cup M | \begin{array}{l} m \in mv(m_1), \\ valid(m, M) \end{array} \right\})) & : \quad otherwise \end{cases} \tag{14.65}$$

其中函数 $valid(m, M)$ 检查新的移动结果 $m = (A'', B'')$ 是否不存在冲突。它要求 A'' 和 B'' 即不能是 3，也不能是 6，并且 m 以前没有尝试过，它不存在于 M 中，

以避免重复的尝试。

$$valid(m, M) = A'' \neq 3, A'' \neq 6, B'' \neq 3, B'' \neq 6, m \notin M \tag{14.66}$$

下面的 Haskell 例子程序实现了狼、羊、白菜问题的解法。为了简单，这里我们使用了普通的列表来表示队列。严格来说应该使用前面章节介绍过的纯函数式队列。

```
import Data.Bits

solve = bfsSolve [[(15, 0)]] where
    bfsSolve :: [[(Int, Int)]] → [(Int, Int)]
    bfsSolve [] = [] -- 无解
    bfsSolve (c:cs) | (fst $ head c) == 0 = reverse c
                    | otherwise = bfsSolve (cs ++ map (:c)
                                        (filter (`valid` c) $ moves $ head c))
    valid (a, b) r = not $ or [ a `elem` [3, 6], b `elem` [3, 6],
                                (a, b) `elem` r]

moves (a, b) = if b < 8 then trans a b else map swap (trans b a) where
    trans x y = [(x - 8 - i, y + 8 + i)
                    | i ←[0, 1, 2, 4], i == 0 || (x ∘ &. i) ≠ 0]
    swap (x, y) = (y, x)
```

可以对这一算法稍作改动，找出所有可能的解，而不是在找出最快的解后结束。作为练习，读者可以尝试这一改动。下面给出了狼、羊、白菜问题的两个最优解。

第一个解：

左岸	河	右岸
狼、羊、白菜、农夫		
狼、白菜		羊、农夫
狼、白菜、农夫		羊
白菜		狼、羊、农夫
羊、百次、农夫		狼
羊		狼、白菜、农夫
羊、农夫		狼、白菜
		狼、羊、白菜、农夫

第二个解：

左岸	河	右岸
狼、羊、白菜、农夫		
狼、白菜		羊、农夫
狼、白菜、农夫		羊
狼		羊、白菜、农夫
狼、羊、农夫		白菜
羊		狼、白菜、农夫
羊、农夫		狼、白菜
		狼、羊、白菜、农夫

这一问题也可以用命令式的方式解决。观察可以发现我们的解是尾递归的，我们可以将它直接转换为循环。我们使用列表 S 来记录所有找到的解。一开始把只含有一个元素的列表 $\{(15,0)\}$ 放入队列。只要队列不为空，我们就调用过程 DeQ 从头部取出元素 C。检查是否到达了最终的目标状态，如果没有，就展开所有可能的移动选项，并将它们加入回队列的尾部，以便后继的搜索。

```
1: function Solve
2:     S ← φ
3:     Q ← φ
4:     EnQ(Q, {(15, 0)})
5:     while Q ≠ φ do
6:         C ← DeQ(Q)
7:         if c_1 = (0, 15) then
8:             Add(S, Reverse(C))
9:         else
10:            for ∀m ∈ Moves(C) do
11:                if Valid(m, C) then
12:                    EnQ(Q, {m} ∪ C)
13:    return S
```

其中过程 Moves 和 Valid 的定义与此前相同。下面的 Python 例子程序实现了狼、羊、白菜问题的解法。

```
def solve():
    s = []
    queue = [[(0xf, 0)]]
    while queue ≠ []:
        cur = queue.pop(0)
        if cur[0] == (0, 0xf):
            s.append(reverse(cur))
        else:
            for m in moves(cur):
                queue.append([m]+cur)
    return s

def moves(s):
    (a, b) = s[0]
    return valid(s, trans(a, b) if b < 8 else swaps(trans(b, a)))

def valid(s, mv):
    return [(a, b) for (a, b) in mv
        if a not in [3, 6] and b not in [3, 6] and (a, b) not in s]

def trans(a, b):
    masks = [ 8 | (1<<i) for i in range(4)]
    return [(a ^ mask, b | mask) for mask in masks if a & mask == mask]
```

```
def swaps(s):
    return [(b, a) for (a, b) in s]
```

这一程序和前面的算法描述略有不同，它在产生可能的移动选项时，同时去掉了含有冲突的情况。

每次农夫渡河时，他都有 m 个可能的选择，其中 m 是农夫所在的河岸上事物的数目。m 总小于 4，因此算法在第 n 次渡河时的运行时间不会超过 n^4。这一估计远远超过实际的时间，我们避免尝试所有含有冲突或重复的情况。最坏情况下，我们的算法会检查所有可能到达的状态。由于需要检查记录以避免重复，算法大约使用 $O(n^2)$ 的时间来搜索第 n 次渡河时的所有可能状态。

倒水问题

倒水问题是一道经典人工智能中的著名趣题。这一问题的历史悠久。只有两个水瓶，一个的容量是 9 升水，另一个的容量是 4 升水。问如何才能从河中取出 6 升水?

这道题目有很多变化形式，瓶子的容积和要取出的水的容量可以是其他数值。有一个故事说解决这道题目的主人公是少年时代的法国数学家和科学家帕斯卡（Blaise Pascal），另一故事说是泊松（Simèon Denis Poisson）。在著名的好莱坞电影《虎胆龙威 3》（Die-Hard 3）中，电影明星布鲁斯，威利斯（Bruce Willis）和塞缪尔，杰克逊（Samuel L. Jackson）也遇到了同样的趣题。

著名的数学家波利亚（Pòlya）在《如何解题》中给出了一个倒推法的解 [90]。

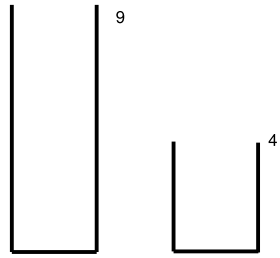


图 14.38: 两个瓶子的容积分别为 9 和 4

从图14.38的起始状态思考会比较困难。波利亚指出，最终的状态是，大瓶子中盛有 6 升水。这样我们可以得知，前一步时，我们从 9 升的大瓶子中倒出 3 升水。为了达成这一点，小瓶子中需要盛有 1 升水。如图14.39所示。

很容易看出，只要倒满 9 升的瓶子，然后连续两次倒入 4 升的瓶子，并将 4 升的瓶子倒空，就可以得到 1 升水。如图14.40所示。此时，我们已经找到解了。通过倒推法，我们可以比较容易地得到 6 升水的获取方法。

波利亚的方法是一种策略性的通用方法。但是仍然无法直接从中得到具体的算法。例如怎样从 899 升和 1147 升的瓶子得到 2 升水?

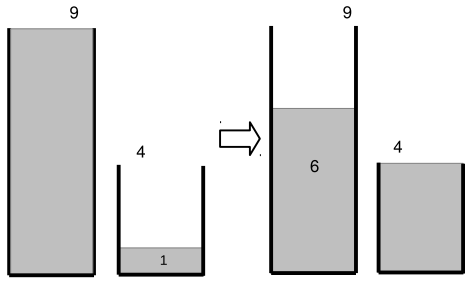


图 14.39: 最后两步

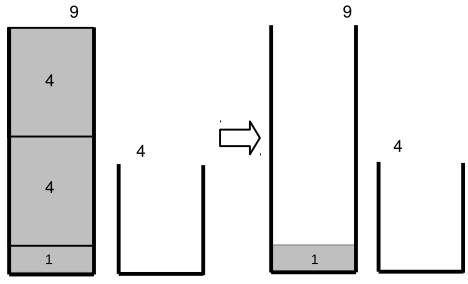


图 14.40: 将大瓶倒满，然后倒入小瓶两次

使用两个瓶子，每次有 6 种操作方法。记小瓶子为 A，大瓶子为 B：

- 将小瓶子 A 装满水；
- 将大瓶子 B 装满水；
- 将小瓶子 A 中的水倒空；
- 将大瓶子 B 中的水倒空；
- 将小瓶子 A 中的水倒入大瓶子 B；
- 将大瓶子 B 中的水倒入小瓶子 A。

下面的是一系列倒水的动作，这里我们假设容积 $a < b < 2a$。

A	B	操作
0	0	开始
a	0	倒满 A
0	a	将 A 倒入 B
a	a	倒满 A
2a - b	b	将 A 倒入 B
2a - b	0	倒光 B
0	2a - b	将 A 倒入 B
a	2a - b	倒满 A
3a - 2b	b	将 A 倒入 B
...	...	...

表 14.8: 两个瓶子内的水量和倒水操作的对应关系

无论进行何种操作，每个瓶子中的水的容量总可以表示为 $xa + yb$ 的形式，其中 a 和 b 分别是两个瓶子的容量，x 和 y 是整数。也就是说，我们能获得的水的体积总是 a 与 b 的线性组合。于是我们立即可以知道，给定两个瓶子的容量，是否可以得到 g 升的水。

例如，使用两个分别容量为 4 升和 6 升的瓶子，我们永远无法得到 5 升的水。通过使用数论中的定理可知，使用两个瓶子，当且仅当 g 能够被瓶子容积的最大公约数整除时，才能得到 g 升水。即：

$$gcd(a, b)|g \tag{14.67}$$

其中 '|' 是整除符号，$m|n$ 表示整数 n 可以被 m 整除。进一步说，如果 a 和 b 互素，即 $gcd(a, b) = 1$，则可以得到任意自然数 g 升水。

虽然通过检查 $gcd(a, b)$ 是否整除 g 可以判断问题是否有解，但是我们并不知道解的具体倒水操作顺序。如果我们可以找到整数 x 和 y，使得 $g = xa + yb$。就可以

得到一组操作（尽管可能不是最优解）来解决此题。具体思路是这样的：不失一般性，设 $x > 0, y < 0$，我们需要倒满瓶子 A 总共 x 次，倒空瓶子 B 总共 y 次。

例如，若小瓶容积 $a = 3$、大瓶容积 $b = 5$，要取得 $g = 4$ 升水，因为 $4 = 3 \times 3 - 5$，我们可以设计下面的一系列操作：

A	B	操作
0	0	开始
3	0	倒满 A
0	3	将 A 倒入 B
3	3	倒满 A
1	5	将 A 倒入 B
1	0	将 B 倒空
0	1	将 A 倒入 B
3	1	倒满 A
0	4	将 A 倒入 B

表 14.9: 取得 4 升水需要进行的操作

在这一系列操作中，我们倒满 A 共 3 次，倒空 B 共 1 次。这一过程可以描述如下。

重复 x 次：

1. 倒满 A；

2. 将 A 倒入 B，若 B 变满，则将其倒空。

因此剩下的唯一问题是寻找整数 x 和 y。数论中有一个强大的工具叫做扩展欧几里得算法（Extended Euclid algorithm），可以用来解决这个问题。经典的欧几里得算法，只能找到最大公约数，而扩展欧几里得算法还可以同时得到一对整数 x 和 y，使得：

$$(d, x, y) = gcd_{ext}(a, b) \tag{14.68}$$

其中 $d = gcd(a, b)$ 为最大公约数，而 $ax + by = d$。不失一般性，设 $a < b$，存在商 q 和余数 r 使得：

$$b = aq + r \tag{14.69}$$

因为 d 是公约数，他可以同时整除 a 和 b，因此 d 也可以整除 r。由于 r 小于 a，我们可以通过寻找 a 和 r 的最大公约数来减小问题的规模。

$$(d, x', y') = gcd_{ext}(r, a) \tag{14.70}$$

根据扩展欧几里得算法的定义，其中 $d = x'r + y'a$。将 $b = aq + r$ 转换为 $r = b - aq$ 并替换上式中的 r，可以得到：

$$\begin{aligned} d &= x'(b - aq) + y'a \\ &= (y' - x'q)a + x'b \end{aligned} \tag{14.71}$$

这正好是 a 与 b 的线性组合，于是我们有：

$$\begin{cases} x = y' - x'\dfrac{b}{a} \\ y = x' \end{cases} \tag{14.72}$$

这是一个典型的递归关系。边界条件发生在 $a = 0$ 时。

$$gcd(0, b) = b = 0a + 1b \tag{14.73}$$

综上，扩展欧几里得算法可以定义如下：

$$gcd_{ext}(a, b) = \begin{cases} (b, 0, 1) & : \quad a = 0 \\ (d, y' - x'\dfrac{b}{a}, x') & : \quad otherwise \end{cases} \tag{14.74}$$

其中 d、x'、y' 的定义如式 (14.70)。

倒水问题几乎解决了，但是我们仍需处理两个具体的问题。第一、扩展欧几里得算法给出了最大公约数及其线性组合。但要取水的容量 g 可能不等于 d，而是 d 的倍数。若 $m = g/gcd(a, b)$，我们可以分别将 x 和 y 乘以 m 倍；第二、我们假设 $x > 0$，来设计了倒满瓶子 A 总共 x 的过程。但扩展欧几里得算法并不保证 x 为正数。例如 $gcd_{ext}(4, 9) = (1, -2, 1)$。若 x 为负数，由于 $d = xa + yb$，我们可以不断将 x 加 b，同时将 y 减 a，直到 x 大于 0。

至此，我们已可以给出完整的两瓶倒水问题的解了。下面的 Haskell 例子程序实现了这一解法。

```
extGcd 0 b = (b, 0, 1)
extGcd a b = let (d, x', y') = extGcd (b `mod` a) a in
               (d, y' - x' * (b `div` a), x')

solve a b g | g `mod` d ≠ 0 = [] -- 无解
            | otherwise = solve' (x * g `div` d)
    where
      (d, x, y) = extGcd a b
      solve' x | x < 0 = solve' (x + b)
               | otherwise = pour x [(0, 0)]
      pour 0 ps = reverse ((0, g):ps)
      pour x ps@((a', b'):_) | a' == 0 = pour (x - 1) ((a, b'):ps) -- fill a
                             | b' == b = pour x ((a', 0):ps) -- empty b
                             | otherwise = pour x ((max 0 (a' + b' - b),
                                                    min (a' + b') b):ps)
```

虽然我们可以用扩展欧几里得算法解决两瓶倒水问题，但是得到的解并不一定是最优的。例如，使用 3 升和 5 升的瓶子，获取 4 升水的时候，扩展欧几里得算法给出如下的操作顺序：

```
[(0,0),(3,0),(0,3),(3,3),(1,5),(1,0),(0,1),(3,1),
(0,4),(3,4),(2,5),(2,0),(0,2),(3,2),(0,5),(3,5),
(3,0),(0,3),(3,3),(1,5),(1,0),(0,1),(3,1),(0,4)]
```

总共需要 23 步，而最优解只需要 6 步：

```
[(0,0),(0,5),(3,2),(0,2),(2,0),(2,5),(3,4)]
```

观察 23 步的解，我们发现在第 8 步时，瓶子 B 中已有 4 升水了。但是算法仍然继续执行后面的 15 步。原因是我们通过扩展欧几里得算法得到的线性组合 x 和 y 并非满足条件的唯一线性组合。在所有满足 $g = xa + by$ 的整数中，$|x| + |y|$ 越小，所需步骤越少。本章附带的练习中有一道题目要求寻找最优的线性组合。

如何寻找最优解？我们有两种策略，一种是寻找 x 和 y，使得 $|x| + |y|$ 最小；另外一种是采用“狼、羊、白菜问题”的思路。本节我们介绍后一种方法。由于我们最多有 6 种可能的操作：倒满 A、倒满 B、将 A 倒入 B、将 B 倒入 A、倒空 A、和倒空 B，我们可以并行尝试所有的操作，检查那个操作可以得到最优解。我们需要记录所有已经到达的状态以避免重复。为了用有限的资源获得并行的效果，我们使用一个队列来安排所有的尝试。队列中保存的元素是一系列值对 (p, q)，其中 p 和 q 分别是两个瓶中盛水的体积。这些值对记录了从开始到最后进行的倒水操作。队列一开始时，唯一的元素是一个列表。表中含有一对值 $\{(0, 0)\}$。

$$solve(a, b, g) = solve'\{\{(0, 0)\}\} \tag{14.75}$$

只要队列不为空，我们就从队列头部取出一个操作序列，如果这一序列中的最后一个状态，包含目标容量 g 升水，则我们找到了一个解，我们将这一序列逆序输出；否则，我们扩展最后一个状态，尝试所有 6 种可能，去掉重复的状态，并将它们加入到队列尾部。记队列为 Q，队列头部保存的序列为 S，S 中最后一对值为 (p, q)，剩下的其余对为 S'。头部元素出队后，队列变为 Q'。这一搜索算法可定义如下：

$$solve'(Q) = \begin{cases} \phi & : \quad Q = \phi \\ reverse(S) & : \quad p = g \vee q = g \\ solve'(EnQ'(Q', \{\{s'\} \cup S' | s' \in try(S)\})) & : \quad otherwise \end{cases} \tag{14.76}$$

其中函数 EnQ' 逐一将列表中的序列加入到队尾。函数 $try(S)$ 尝试所有 6 种操作，并产生新的水的体积对：

$$try(S) = \{s' | s' \in \begin{Bmatrix} fillA(p, q), fillB(p, q), \\ pourA(p, q), pourB(p, q), \\ emptyA(p, q), emptyB(p, q) \end{Bmatrix}, s' \notin S'\} \tag{14.77}$$

6 种操作的定义很直观。对于倒满操作，结果是水瓶中水的体积达到瓶子的容积；对于倒空操作，瓶中水的体积为 0；对于倒入操作，我们需要检查目标瓶子的剩余容量是否足够大。

$$\begin{aligned} fillA(p,q) &= (a,q) & fillB(p,q) &= (p,b) \\ emptyA(p,q) &= (0,q) & emptyB(p,q) &= (p,0) \\ pourA(p,q) &= (max(0,p+q-b), min(x+y,b)) & & \\ pourB(p,q) &= (min(x+y,a), max(0,x+y-a)) & & \end{aligned} \tag{14.78}$$

下面的 Haskell 程序实现了这一解法。

```
solve' a b g = bfs [[(0, 0)]] where
  bfs [] = []
  bfs (c:cs) | fst (head c) == g || snd (head c) == g = reverse c
             | otherwise = bfs (cs ++ map (:c) (expand c))
  expand ((x, y):ps) = filter (`notElem` ps) $ map (λf → f x y)
                              [fillA, fillB, pourA, pourB, emptyA, emptyB]
  fillA _ y = (a, y)
  fillB x _ = (x, b)
  emptyA _ y = (0, y)
  emptyB x _ = (x, 0)
  pourA x y = (max 0 (x + y - b), min (x + y) b)
  pourB x y = (min (x + y) a, max 0 (x + y - a))
```

这一方法总返回最快的解法。它也可以用命令式的方法实现。我们无需在队列的每个元素中保存全部的操作序列，可以建立一个全局的历史记录列表，然后使用指针链接操作的顺序。这样能节省大量的空间。

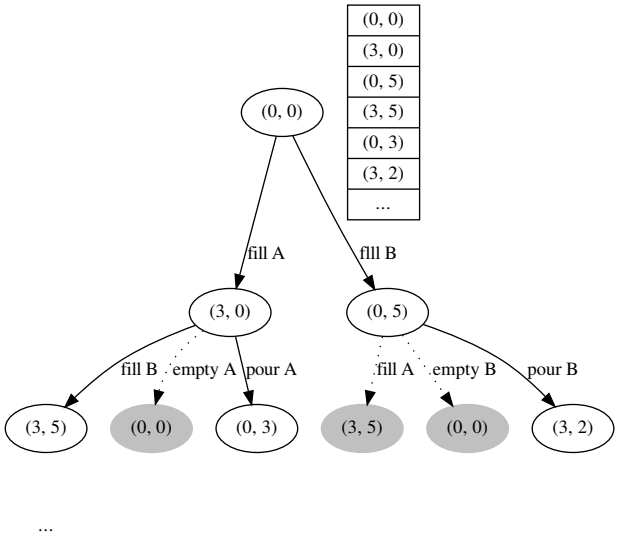


图 14.41: 所有尝试过的状态都存储于一个全局的列表中

如图14.41所示，初始状态为 (0, 0)。只有 ‘fillA’ 和 ‘fillB’ 可行。它们被加入记录；接下来，我们在记录的结果 (3, 0) 的基础上尝试 ‘fillB’，并将新结果 (3, 5) 记录下来。

但是在 (3, 0) 的基础上尝试 'empty A' 将回到初始状态 (0, 0)。由于我们已记录了这一状态，所以这一选项被跳过。图中，所有灰色的状态，都是重复状态。

通过这样的设计，我们无需在队列的每个元素中记录操作的序列。我们可以给图14.41中的每个节点增加一个父节点指针，并用它从任意状态回溯到初始状态。下面的 C 语言例子代码给出了这一设计的定义。

```
struct Step {
    int p, q;
    struct Step* parent;
};

struct Step* make_step(int p, int q, struct Step* parent) {
    struct Step* s = (struct Step*) malloc(sizeof(struct Step));
    s->p = p;
    s->q = q;
    s->parent = parent;
    return s;
}
```

其中 p 和 q 是两个水瓶中盛水的体积。对于任何状态 s，定义函数 $p(s)$ 和 $q(s)$ 分别返回这两个量，命令式算法可以实现如下：

```
1:  function SOLVE(a, b, g)
2:      Q ← φ
3:      PUSH-AND-RECORD(Q, (0, 0))
4:      while Q ≠ φ do
5:          s ← POP(Q)
6:          if p(s) = g ∨ q(s) = g then
7:              return s
8:          else
9:              C ← EXPAND(s)
10:             for ∀c ∈ C do
11:                 if c ≠ s ∧ ¬ VISITED(c) then
12:                     PUSH-AND-RECORD(Q, c)
13:     return NIL
```

其中 PUSH-AND-RECORD 不仅将元素加入队列尾部，还将其记录入访问过的状态的表中，这样将来就可以检查是否到达过此状态。所有的 push 操作都将新元素加入到列表的尾部。对于 pop 操作，我们并不将元素删除，而是将头指针向后移动一步。这一包含所有历史数据的列表必须在使用前清空。下面的 C 语言例子程序实现了这一算法。

```
struct Step *steps[1000], **head, **tail = steps;

void push(struct Step* s) { *tail++ = s; }
```

```
struct Step* pop() { return *head++; }

int empty() { return head == tail; }

void reset() {
    struct Step **p;
    for (p = steps; p ≠ tail; ++p)
        free(*p);
    head = tail = steps;
}
```

为了检查一个状态是否访问过，我们需要遍历列表，比较 p 和 q 的值。

```
int eq(struct Step* a, struct Step* b) {
    return a→p == b→p && a→q == b→q;
}

int visited(struct Step* s) {
    struct Step **p;
    for (p = steps; p ≠ tail; ++p)
        if (eq(*p, s)) return 1;
    return 0;
}
```

主程序实现如下：

```
struct Step* solve(int a, int b, int g) {
    int i;
    struct Step *cur, *cs[6];
    reset();
    push(make_step(0, 0, NULL));
    while (!empty()) {
        cur = pop();
        if (cur→p == g || cur→q == g)
            return cur;
        else {
            expand(cur, a, b, cs);
            for (i = 0; i < 6; ++i)
                if(!eq(cur, cs[i]) && !visited(cs[i]))
                    push(cs[i]);
        }
    }
    return NULL;
}
```

其中函数 expand 尝试所有 6 种操作：

```
void expand(struct Step* s, int a, int b, struct Step** cs) {
    int p = s→p, q = s→q;
    cs[0] = make_step(a, q, s); /*fillA*/
    cs[1] = make_step(p, b, s); /*fillB*/
    cs[2] = make_step(0, q, s); /*emptyA*/
    cs[3] = make_step(p, 0, s); /*emptyB*/
    cs[4] = make_step(max(0, p + q - b), min(p + q, b), s); /*pourA*/
```

```
    cs[5] = make_step(min(p + q, a), max(0, p + q - a), s); /*pourB*/
}
```

结果步骤可以通过父指针不断向上逆序输出，如下面的递归函数实现：

```
void print(struct Step* s) {
    if (s) {
        print(s→parent);
        printf("%d, %dλn", s→p, s→q);
    }
}
```

华容道

华容道是一种滑块类游戏，国外称 Kloski。在很多国家都有类似的游戏。滑块的大小和布局会有不同。图14.42是中国传统的华容道游戏。

(a) 起始布局 (b) 移动若干步后的样子

图 14.42: 华容道游戏

华容道游戏中，共有 10 个滑块，上面标有数字或者图案。最小的滑块大小为一个单位的正方形，最大的一块为 2×2 单位。在棋盘下方的中间，有一个宽度为 2 个单位长的缺口。最大的一块代表曹操，其他的为刘备手下的五虎上将和士兵。游戏的目标是要通过滑动，将曹操移动到棋盘最下方逃走。图14.43是日本的类似游戏，名叫“箱子中的女儿”，最大的一块代表女儿，剩余滑块代表其他家庭成员。

本节中，我们要找出一种解法，通过一系列移动，用最少的步数，将滑块从初始状态，变换到目标状态。

最直观的想法，是用一个 5×4 矩阵来代表棋盘。每个棋子被标记为一个数字。下面的矩阵 M，给出了华容道的初始状态。

图 14.43: 日本的“箱子中的女儿”游戏

$$M = \begin{bmatrix} 1 & 10 & 10 & 2 \\ 1 & 10 & 10 & 2 \\ 3 & 4 & 4 & 5 \\ 3 & 7 & 8 & 5 \\ 6 & 0 & 0 & 9 \end{bmatrix}$$

在矩阵中，值为 i 的元素表示相应的位置被第 i 个棋子所占。特殊值 0 代表空位置。通过使用序列 1、2、……来代表棋子，一个布局可以进一步用一个数组 L 来代表。每个元素是一个列表，包含若干被该元素所代表的棋子覆盖的所有位置。例如 $L[4] = \{(3,2),(3,3)\}$ 表示，第 4 个棋子覆盖了位置 $(3,2)$ 和 $(3,3)$，其中 (i,j) 表示在第 i 行、第 j 列的位置。

华容道的初始布局可以用这种方法写成下面的数组。

$$\{\{(1,1),(2,1)\},\{(1,4),(2,4)\},\{(3,1),(4,1)\},\{(3,2),(3,3)\},\{(3,4),(4,4)\},$$
$$\{(5,1)\},\{(4,2)\},\{(4,3)\},\{(5,4)\},\{(1,2),(1,3),(2,2),(2,3)\}\}$$

解华容道时，我们需要检查全部 10 个棋子，看看能否在上下左右 4 个方向移动。看起来这是一个巨大的解空间，每步都有 10×4 个选项，走 n 步后，会有 40^n 种情况。但实际上的情况没有这么多。例如在第一步的时候，只有 4 种可能：将第 6 块向右移动；将第 7 块或第 8 块向下移动；以及将第 9 块向左移动。所有其它选项都不可能发生。图14.44给出了检查某种移动是否可行的方法。

左侧的例子描述了将标为 1 的棋子向下滑动一个单位的情况。这个棋子覆盖两个格子。上方的 1 要移动到的格子此前也被这个棋子所占，所以格子的值也为 1；下方的 1 要移动到一个空格子，空格子标记为 0；

右侧的例子描述了一个不可行的移动。这个例子中，虽然棋子上方的部分可以移动到一个被同样棋子所占的格子中，但是下方部分的 1 不能移动到被其它棋子 2 所占的格子中。

为了确定一个移动是否合法，我们需要检查棋子覆盖的所有格子将要移动到的位置，如果目标位置的格子为 0，或者数字相同，移动就是可行的。否则就会和其它棋子冲突。对于布局 L，对应的矩阵为 M，设我们要移动第 k 个棋子，移动方向为

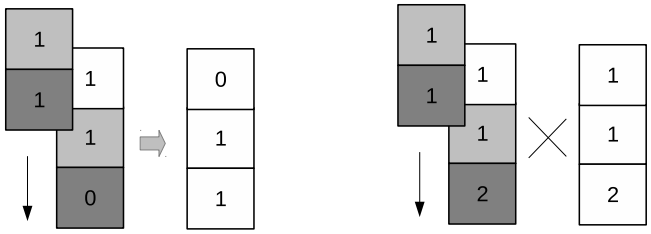


图 14.44: 左侧：两个标为 1 的格子都可以移动；右侧：上方标为 1 的格子虽然可以，但是下方标为 1 的格子和标为 2 的格子冲突。

$(\Delta x, \Delta y)$，其中 $|\Delta x| \leq 1$、$|\Delta y| \leq 1$。下面的等式，定义了移动是否可行：

$$\begin{aligned} & valid(L, k, \Delta x, \Delta y): \\ & \quad \forall (i,j) \in L[k] \Rightarrow \quad i' = i + \Delta y, j' = j + \Delta x, \\ & \qquad\qquad\qquad\quad (1,1) \leq (i', j') \leq (5,4), M_{i'j'} \in \{k, 0\} \end{aligned} \tag{14.79}$$

解决华容道问题的另一个重点是如何避免重复的尝试。经过一系列的移动，我们可能会回到此前的某个布局。但是，仅仅避免出现相同的矩阵是不够的，考虑下面给出的两个矩阵，虽然 $M_1 \neq M_2$，但是我们仍然要避免移动到 M_2，因为他们本质上是相同的。

$$M_1 = \begin{bmatrix} 1 & 10 & 10 & 2 \\ 1 & 10 & 10 & 2 \\ 3 & 4 & 4 & 5 \\ 3 & 7 & 8 & 5 \\ 6 & 0 & 0 & 9 \end{bmatrix} \quad M_2 = \begin{bmatrix} 2 & 10 & 10 & 1 \\ 2 & 10 & 10 & 1 \\ 3 & 4 & 4 & 5 \\ 3 & 7 & 6 & 5 \\ 8 & 0 & 0 & 9 \end{bmatrix}$$

这一事实告诉我们，需要比较布局，而不仅仅是矩阵来避免出现重复。记上述矩阵对应的布局分别为 L_1 和 L_2，可以很容易验证 $||L_1|| = ||L_2||$，其中 $||L||$ 是归一化的布局，其定义如下：

$$||L|| = sort(\{sort(l_i) | \forall l_i \in L\}) \tag{14.80}$$

归一化的布局中，所有的元素都排好序，并且每个元素内部也都是有序的。相互间的顺序定义为：$(a, b) \leq (c, d) \Leftrightarrow an + b \leq cn + d$，其中 n 是矩阵的宽度。

观察到华容道的棋盘是对称的，因此布局也可以有对称布局。出现对称的布局也是一种重复，我们需要避免它。例如下面的 M_1 和 M_2 就是对称的布局。

$$M_1 = \begin{bmatrix} 10 & 10 & 1 & 2 \\ 10 & 10 & 1 & 2 \\ 3 & 5 & 4 & 4 \\ 3 & 5 & 8 & 9 \\ 6 & 7 & 0 & 0 \end{bmatrix} \quad M_2 = \begin{bmatrix} 3 & 1 & 10 & 10 \\ 3 & 1 & 10 & 10 \\ 4 & 4 & 2 & 5 \\ 7 & 6 & 2 & 5 \\ 0 & 0 & 9 & 8 \end{bmatrix}$$

注意到它们的归一化布局也是相互对称的。通过下面方法可以很容易得到一个对称的布局。

$$mirror(L) = \{\{(i, n - j + 1) | \forall (i, j) \in l\} | \forall l \in L\} \tag{14.81}$$

我们发现矩阵对于验证移动是否可行很方便，而布局形式便于表达移动和避免重复。我们可以用类似的方法来解决华容道游戏。使用一个队列，队列中的每个元素包含两部分：一系列移动，和这些移动导致的布局。每次移动的形式为 $(k, (\Delta y, \Delta x))$，表示在棋盘上移动第 k 个棋子，移动方向为 $(\Delta x, \Delta y)$。

最开始的时候，队列中包含起始布局。只要队列不为空，我们就从队列头部取出一个元素，检查最大的一块棋子是否已经到达目标位置，即 $L[10] = \{(4, 2), (4, 3), (5, 2), (5, 3)\}$。如果到达，则结束；否则，我们对每块棋子尝试向上下左右 4 个方向移动，并把所有可行的、不重复的布局存入队列尾部。在整个搜索过程中，我们需要保存所有找到的归一化布局以避免重复。

记队列为 Q，布局的历史记录为 H，队列头部记录的第一个布局为 L，它对应的矩阵为 M。到这个布局为止的一系列移动为 S。下面的算法定义了华容道游戏的解法。

$$solve(Q, H) = \begin{cases} \phi & : \quad Q = \phi \\ reverse(S) & : \quad L[10] = \{(4, 2), (4, 3), (5, 2), (5, 3)\} \\ solve(Q', H') & : \quad otherwise \end{cases} \tag{14.82}$$

第一行表示，如果队列为空，我们已经尝试了所有可能的移动方案，但是未能找到可行的解；第二行表示我们找到了一个解，我们将移动序列逆序返回；这两种是边界情况。否则，算法从当前的布局扩展出所有可行的移动方案，并将新布局加入到队列的尾部。新队列记为 Q'，更新后的布局历史记录为 H'。然后程序进行递归搜索。

为了将一个布局扩展为不重复的新布局，我们定义了如下的函数：

$$\begin{aligned} expand(L, H) = \{(k, (\Delta y, \Delta x) | \quad & \forall k \in \{1, 2, ..., 10\}, \\ & \forall (\Delta y, \Delta x) \in \{(0, -1), (0, 1), (-1, 0), (1, 0)\}, \\ & valid(L, k, \Delta x, \Delta y), unique(L', H)\} \end{aligned} \tag{14.83}$$

其中 L' 是将布局 L 中的第 k 块棋子移动 $(\Delta y, \Delta x)$ 后得到的新布局，M' 是新布局对应的矩阵，M'' 是 L' 的对称布局所对应的矩阵。函数 $unique$ 定义如下：

$$unique(L', H) = M' \notin H \wedge M'' \notin H \tag{14.84}$$

由于纯函数环境中无法更改数组的内容，我们使用基于树的 map 来代表布局[11]。下面的 Haskell 例子程序定义了一些类型名称。

```
import qualified Data.Map as M
import Data.Ix
import Data.List (sort)

type Point = (Integer, Integer)
type Layout = M.Map Integer [Point]
type Move = (Integer, Point)

data Ops = Op Layout [Move]
```

主程序和上面定义的 $solve(Q, H)$ 类似。

```
solve :: [Ops] → [[[Point]]]→ [Move]
solve [] _ = [] -- 无解
solve (Op x seq : cs) visit
    | M.lookup 10 x == Just [(4, 2), (4, 3), (5, 2), (5, 3)] = reverse seq
    | otherwise = solve q visit'
  where
    ops = expand x visit
    visit' = map (layout ∘ move x) ops ++ visit
    q = cs ++ [Op (move x op) (op:seq) | op ← ops ]
```

其中函数 `layout` 通过排序给出归一化的布局。函数 `move` 通过滑动第 i 块棋子 $(\Delta y, \Delta x)$ 距离得到新的 map。

```
layout = sort ∘ map sort ∘ M.elems

move x (i, d) = M.update (Just ∘ map (flip shift d)) i x

shift (y, x) (dy, dx) = (y + dy, x + dx)
```

函数 `expand` 返回所有可行的移动方案，如前面的 $expand(L, H)$ 定义所示。

```
expand :: Layout → [[[Point]]] → [Move]
expand x visit = [(i, d) | i ←[1..10],
                           d ← [(0, -1), (0, 1), (-1, 0), (1, 0)],
                           valid i d, unique i d] where
  valid i d = all (λp → let p' = shift p d in
                    inRange (bounds board) p' &&
                    (M.keys $ M.filter (elem p') x) `elem` [[i], []])
              (maybe [] id $ M.lookup i x)
  unique i d = let mv = move x (i, d) in
               all (`notElem` visit) (map layout [mv, mirror mv])
```

[11] 也可以使用前面章节定义的手指树。

我们需要去掉对称的布局，函数 `mirror` 的定义如下：

```
mirror = M.map (map (λ (y, x) → (y, 5 - x)))
```

这一程序需要数分钟产生华容道“横刀立马”布局的最优解，总共需要 116 步，最后 3 步如下：

```
...

['5', '3', '2', '1']
['5', '3', '2', '1']
['7', '9', '4', '4']
['A', 'A', '6', '0']
['A', 'A', '0', '8']

['5', '3', '2', '1']
['5', '3', '2', '1']
['7', '9', '4', '4']
['A', 'A', '0', '6']
['A', 'A', '0', '8']

['5', '3', '2', '1']
['5', '3', '2', '1']
['7', '9', '4', '4']
['0', 'A', 'A', '6']
['0', 'A', 'A', '8']

total 116 steps
```

也可以用命令式的方法实现华容道的解法。注意到 $solve(Q, H)$ 是尾递归的，它可以很容易地翻译为循环。我们可以将每个布局链接到它的父布局上，这样就可以在全局范围内记录移动的顺序。使用这种方法可以节省空间，队列中的每个元素无需再记录移动顺序的信息。当输出结果的时候，我们只要从最终结果沿着父布局指针向上回溯即可。

令函数 $\textsc{Link}(L', L)$ 将新布局 L' 链接到它的父布局 L 上。下面的算法接受一个起始布局，然后搜索最佳解法。

1: **function** $\textsc{Solve}(L_0)$
2: $H \leftarrow ||L_0||$
3: $Q \leftarrow \phi$
4: $\textsc{Push}(Q, \textsc{Link}(L_0, \text{NIL}))$

```
    while Q ≠ φ do
        L ← POP(Q)
        if L[10] = {(4,2),(4,3),(5,2),(5,3)} then
            return L
        else
            for each L' ∈ EXPAND(L, H) do
                PUSH(Q, LINK(L', L))
                APPEND(H, ||L'||)
    return NIL                                  ▷ 无解
```

下面的 Python 例子程序实现了这一解法。

```
class Node:
    def __init__(self, l, p = None):
        self.layout = l
        self.parent = p

def solve(start):
    visit = set([normalize(start)])
    queue = deque([Node(start)])
    while queue:
        cur = queue.popleft()
        layout = cur.layout
        if layout[-1] == [(4, 2), (4, 3), (5, 2), (5, 3)]:
            return cur
        else:
            for brd in expand(layout, visit):
                queue.append(Node(brd, cur))
                visit.add(normalize(brd))
    return None #no solution
```

其中 normalize 和 expand 实现如下:

```
def normalize(layout):
    return tuple(sorted([tuple(sorted(r)) for r in layout]))

def expand(layout, visit):
    def bound(y, x):
        return 1 <= y and y <= 5 and 1 <= x and x <= 4
    def valid(m, i, y, x):
        return m[y - 1][x - 1] in [0, i]
    def unique(brd):
        (m, n) = (normalize(brd), normalize(mirror(brd)))
        return m not in visit and n not in visit
    s = []
    d = [(0, -1), (0, 1), (-1, 0), (1, 0)]
    m = matrix(layout)
    for i in range(1, 11):
        for (dy, dx) in d:
            if all(bound(y + dy, x + dx) and valid(m, i, y + dy, x + dx)
                    for (y, x) in layout[i - 1]):
```

```
                brd = move(layout, (i, (dy, dx)))
                if unique(brd):
                    s.append(brd)
    return s
```

和大多数编程语言一样，Python 中的数组索引从 0 开始，在处理时需要注意。其他函数，包括 mirror、matrix、和 move 的实现如下。

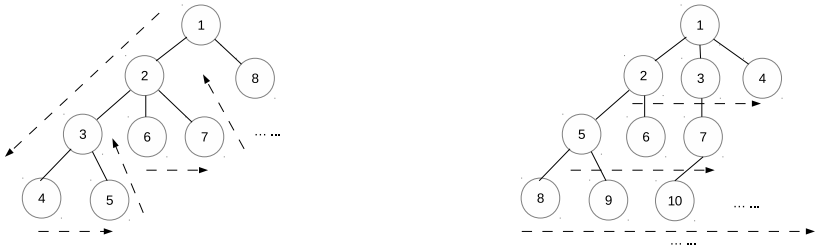
```
def mirror(layout):
    return [[(y, 5 - x) for (y, x) in r] for r in layout]

def matrix(layout):
    m = [[0]*4 for _ in range(5)]
    for (i, ps) in zip(range(1, 11), layout):
        for (y, x) in ps:
            m[y - 1][x - 1] = i
    return m

def move(layout, delta):
    (i, (dy, dx)) = delta
    m = dup(layout)
    m[i - 1] = [(y + dy, x + dx) for (y, x) in m[i - 1]]
    return m

def dup(layout):
    return [r[:] for r in layout]
```

可以修改这一算法，使得它不仅找出华容道的最优解，还能找出所有的可能解法。这种情况下，计算时间和搜索空间 V 成正比，其中 V 包含从起始状态开始可以转换到的所有状态。若将所有这些状态存储在全局空间，并使用父指针将后继状态链接起来，则这一算法的空间复杂度也是 $O(V)$。

广度优先搜索的小结

上述三个问题：狼、羊、和白菜过河问题；倒水问题；和华容道游戏的解有着共同的结构。和深度优先搜索问题类似，它们也都有起始状态和终止状态。在“狼、羊、白菜过河”问题中，起始状态是农夫、狼、羊、和白菜都在河的一岸，而对岸为空；它的终止状态是所有这些都移动到了河对岸。倒水问题的起始状态，两个瓶子都为空，而终止状态是其中任何一个瓶子盛有指定容量的水。华容道问题的起始状态是某种布局（如“横刀立马”），终止状态是另外一个布局，其中最大的棋子移动到了指定的位置。

每个问题都有一系列的规则，可以从一个状态转移到另外一个状态。和深度优先搜索不同，我们“并行“地尝试所有可能的选项。在同一步内所有选项未被尝试完之前，我们不会进一步深入搜索。这一方法保证了具有最小步骤的解可以在其他解之前找出。对比图14.45可以发现这两种不同的搜索策略之间的差异。由于我们总是向水平方向扩展搜索空间，这种搜索被称为广度优先搜索（BFS）。



(a) 深度优先搜索 (b) 广度优先搜索

图 14.45: 深度优先搜索和广度优先搜索的顺序

由于我们无法真正的“并行”搜索，广度优先搜索通常使用一个队列来保存已作出的尝试。尝试步骤较少的候选项被从队列的头部取出，需要较多步骤的新的候选项被加入的队列的尾部。这里要求支持常数时间的入队和出队操作，我们在前面章节介绍的队列可以符合这一需求。严格讲，上面例子程序中的队列并不满足这一条件。它们使用列表来模拟队列，因此入队操作是线性时间的，而非常数时间。读者可以使用我们前面介绍的纯函数式队列来替换它们。

广度优先搜索提供了一种简单的方法来寻找最少步骤的解，但是它不能直接用来搜索其它的最优解。考虑如图14.46所示的一幅有向图，每段路径的长度不同，我们无法用广度优先搜索来找出两个城市之间的最短路径。

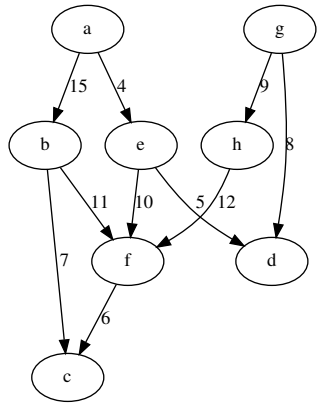


图 14.46: 带权重的有向图

注意从城市 a 到城市 c 之间的最短路径并非经过最少城市的 $a \to b \to c$。这条路径的总长度为 22；而是经过更多城市的路径 $a \to e \to f \to c$，他的总长度只有 20。下一节将介绍搜索最优解的其他方法。

14.3.2 搜索最优解

很多情况下，需要搜索最优解。人们需要“最好”的解来节省时间、空间、成本、或是能量。但是使用有限的资源搜索最优解并不容易。有很多问题的最优解只能通过暴力方法获得。尽管如此，人们发现对于某些特定问题，存在着较简单的方法能够找到最优解。

贪心算法

本节介绍“贪心策略”，也称“贪心算法”。对于某些特定问题，贪心策略能够以较小的代价，相对容易地获得最优解。我们首先介绍信息论中著名的 Huffman 编码问题；然后介绍在特定的币值系统中的换零钱问题。它们都是贪心算法的典型问题。

Huffman 编码

Huffman 编码是一种用最小长度对信息编码的方法。考虑常见的 ASCII 码，它使用 7 个二进制位来对字母、数字、和某些符号编码。ASCII 码可以表达 $2^7 = 128$ 种不同的字符。只使用 0 和 1，我们需要至少 $\log_2 n$ 位来分辨 n 中不同的字符。如果限定只有大写的英文字符，我们可以定义如表14.10所示的的码表。

字符	编码	字符	编码
A	00000	N	01101
B	00001	O	01110
C	00010	P	01111
D	00011	Q	10000
E	00100	R	10001
F	00101	S	10010
G	00110	T	10011
H	00111	U	10100
I	01000	V	10101
J	01001	W	10110
K	01010	X	10111
L	01011	Y	11000
M	01100	Z	11001

表 14.10: 一个大写英文字符的码表

使用这一码表，文本“INTERNATIONAL”可以编码为 65 位的二进制数：

```
00010101101100100100100011011000000110010001001110101100000011010
```

观察上面的码表，它将字母 A 到 Z 映射为 0 到 25 的整数。每个编码使用 5 个二进制位。例如，零被强制使用 5 位，即 00000 而非 0。这样的编码方式被称为“固定长度编码”。

另一种编码方式是“变长编码”。我们可以只用一个二进制位的 0 来代表 A，用两个二进制位的 10 代表 C，用 5 个二进制位的 11001 代表 Z。虽然这种方式可以显著缩短编码总长度。但是在解码的时候，会造成歧义。例如当遇到二进制数 1101，我们不知道它是一个 1，后面跟着一个 101，即字符串“BF”；还是一个 110，后面跟着一个 1，它代表字符串“GB”；或是 1101，它代表字符 N。

著名的摩尔斯电码是变长编码。最常用的字符 E 被编码为一个点，而字符 Z 被编码为两个划和两个点。摩尔斯电码使用特殊的终止符来分割编码，所以不会发生上面的歧义问题。还有其他的方法可以避免歧义，考虑下面的码表：

字符	编码	字符	编码
A	110	E	1110
I	101	L	1111
N	01	O	000
R	001	T	100

表 14.11: 一个无歧义码表

文本“INTERNATIONAL”依照此码表被编码为 38 位的二进制数：

```
10101100111000101110100101000011101111
```

如果按照上述码表解码，我们不会遇到任何有歧义的字符。这是因为没有任何字符的编码是其他编码的前缀。这样的编码称为前缀码（英文为 prefix-code，读者可能会奇怪为何它不叫无前缀码 non-prefix code）。使用前缀码，我们不需要任何分隔符。这样编码的长度就可以缩短。

这自然引发了一个有趣的问题：给定一个文本，我们能否找到一个码表，使得编码长度最短？1951 年，还是 MIT 的一名学生的 David A. Huffman 正好遇到了这个问题 [91]。他的老师 Robert M. Fano 在课上宣布，如果谁解出了这个问题，就不用参加期末考试了。Huffman 尝试了很久，他几乎要放弃了，开始着手准备参加考试。恰在此时，他忽然找到了一个高效的解法。

这一方法的思路是根据字符在文本中出现的频率构造码表。最常用字符的编码最短。

首先可以处理文本，获得每个字符出现的次数。这样我们就有了一个字符集，每个字符都有一个权重。权重为一个表示该字符出现频率的一个数字，它可以是出现的次数，或者是出现的概率。

Huffman 发现，可以使用一棵二叉树来产生前缀码。所有的字符都保存在叶子节点。通过从根节点遍历树产生编码。当向左前进时，我们添加一个 0，向右前进时，添加一个 1。

图14.47描述了一棵二叉树。例如，当我们从根节点出发遍历到 N 时，我们首先向左，然后向右到达 N，因此 N 的编码为 01；而对于字符 A，我们需要向右、向右，再向左。因此 A 的编码是 110。注意，这一方法保证没有任何编码是其它编码的前缀。

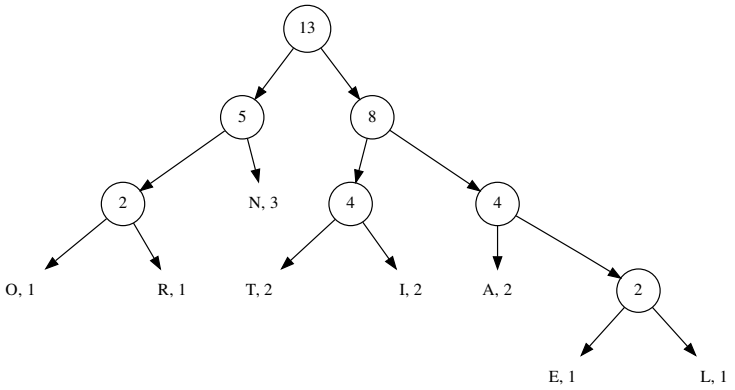


图 14.47: 一棵编码树

这棵树还可以直接用来解码。当扫描一串二进制位时，若某一位为 0，则向左前进；若为 1，则向右前进。当到达叶子节点时，节点上的字符就是解码内容。然后我们重新返回根节点，继续处理剩余的二进制位。

我们需要从一个字符及其权重的列表，构造一棵二进制树，使得最大权重的字符，距离根节点的最近。Huffman 提出了一个自底向上的解法。开始的时候，所有的字符都放入一个叶子节点中。每次我们选出两个权重最小的节点，然后把它们合并成一个分支节点。分支的权重为两个子树的权重和。我们不断选择权重最小的两棵树合并，直到最后得到一棵树。图14.48描述了这一构造过程。

我们可以重用二叉树的定义用于实现 Huffman 编码。每个节点要增加一个权重信息，只有叶子节点保存有字符。下面的 C 语言例子代码定义了这样的节点。

```
struct Node {
    int w;
    char c;
    struct Node *left, *right;
};
```

我们也可以增加一些限制条件，由于树不为空。一棵 Huffman 树要么是一个叶子节点，包含一个字符和它的权重；要么是一个分支节点，记录有它所有叶子节点的权重和。下面的 Haskell 例子代码，定义了这两种情况。

```
data HTr w a = Leaf w a | Branch w (HTr w a) (HTr w a)
```

当合并两棵 Huffman 树 T_1 和 T_2 时，我们建立一个新的分支节点，令这两棵树为新节点的子树。我们可以选择任何一棵作为左子树，另一棵作为右子树。合并结果

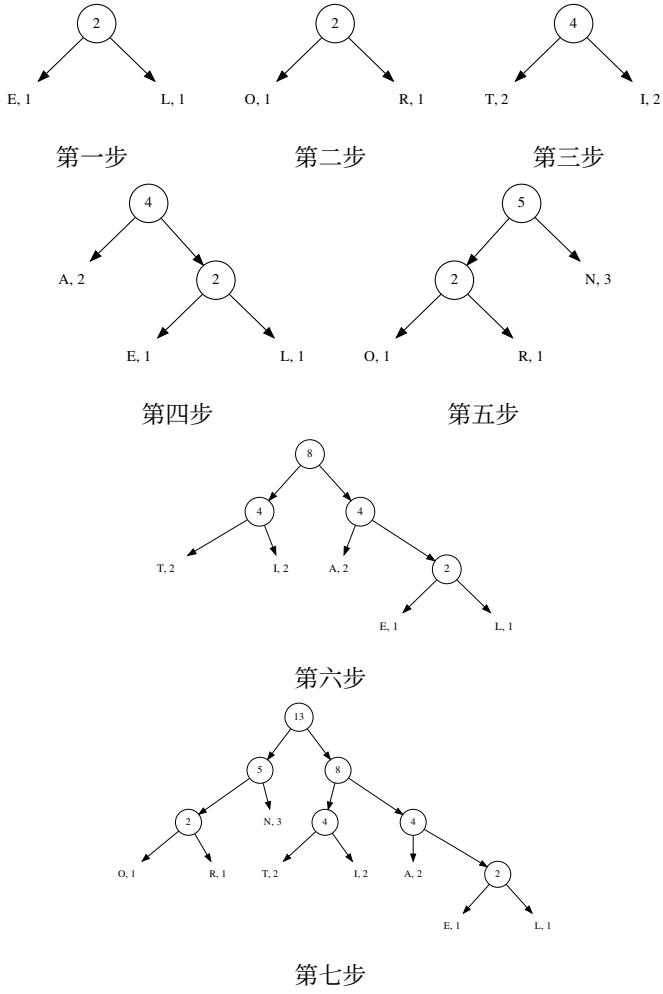


图 14.48: 构造一棵 Haffman 树的步骤

为一棵树 T，他的权重为两棵子树权重的和。即 $w = w_1 + w_2$。若 $w_1 < w_2$，我们定义 $T_1 < T_2$，下面给出了 Huffman 树构造算法的一种定义。

$$build(A) = \begin{cases} T_1 & : & A = \{T_1\} \\ build(\{merge(T_a, T_b)\} \cup A') & : & otherwise \end{cases} \tag{14.85}$$

A 为若干树的列表。它一开始含有所有字符及其权重的叶子节点。若 A 中只有一棵树，则构造结束，这棵树就是最终的 Huffman 树。否则，我们取出权重最小的两棵树 T_a 和 T_b，剩余的树所在的列表为 A'。然后将 T_a 和 T_b 合并为一棵更大的树，并放回列表以进行递归的构造。

$$(T_a, T_b, A') = extract(A) \tag{14.86}$$

我们可以逐一检查所有的树，以找到权重最小的两棵。下面的等式定义了这一过程开始时的情况，比较最前面的两个元素，并作为权重最小的两棵树的候选。同时传入一个空的累积器（accumulator）作为最后一个参数。

$$extract(A) = extract'(min(T_1, T_2), max(T_1, T_2), \{T_3, T_4, ...\}, \phi) \tag{14.87}$$

我们逐一检查剩余的树，若其权重小于两棵最小权重候选树中的任何一棵，我们就修改候选结果，包含这棵树。对于包含树的列表 A，记其中第一棵树为 T_1，除 T_1 外的其余树为 A'。这一扫描过程可以定义如下。

$$extract'(T_a, T_b, A, B) = \begin{cases} (T_a, T_b, B) & : & A = \phi \\ extract'(T_a', T_b', A', \{T_b\} \cup A) & : & T_1 < T_b \\ extract'(T_a, T_b, A', \{T_1\} \cup A) & : & otherwise \end{cases} \tag{14.88}$$

其中 $T_a' = min(T_1, T_a)$、$T_b' = max(T_1, T_a)$ 为更新后的两棵最小权重的树。

下面的 Haskell 例子程序实现了 Huffman 树的构造算法。

```
build [x] = x
build xs = build ((merge x y) : xs') where
  (x, y, xs') = extract xs

extract (x:y:xs) = min2 (min x y) (max x y) xs [] where
  min2 x y [] xs = (x, y, xs)
  min2 x y (z:zs) xs | z < y = min2 (min z x) (max z x) zs (y:xs)
                     | otherwise = min2 x y zs (z:xs)
```

也可以用命令式的方法实现 Huffman 树的构造过程。我们使用一个数组来存储 Huffman 树，最后两个元素是权重最小的树的候选。然后我们从右向左扫描剩余的树，当遇到一个权重更小树，我们就将其和最后两个元素中，权重较大的一个互换。当所有的树都检查完毕后，我们将最后的两棵树合并，并丢弃掉最后一个数组的元素。这样数组的空间就减小 1 个单位。我们重复这一过程直到只剩下最后一棵树。

1: **function** HUFFMAN(A)

```
2:  while |A| > 1 do
3:      n ← |A|
4:      for i ← n − 2 down to 1 do
5:          if A[i] < MAX(A[n], A[n − 1]) then
6:              EXCHANGE A[i] ↔ MAX(A[n], A[n − 1])
7:      A[n − 1] ← MERGE(A[n], A[n − 1])
8:      DROP(A[n])
9:  return A[1]
```

下面的 C++ 例子程序实现了这一算法。在这一程序中，我们不要求最后两棵树已序。

```
typedef vector<Node*> Nodes;

bool lessp(Node* a, Node* b) { return a→w < b→w; }

Node* max(Node* a, Node* b) { return lessp(a, b) ? b : a; }

void swap(Nodes& ts, int i, int j, int k) {
    swap(ts[i], ts[ts[j] < ts[k] ? k : j]);
}

Node* huffman(Nodes ts) {
    int n;
    while((n = ts.size()) > 1) {
        for (int i = n - 3; i ≥ 0; --i)
            if (lessp(ts[i], max(ts[n-1], ts[n-2])))
                swap(ts, i, n-1, n-2);
        ts[n-2] = merge(ts[n-1], ts[n-2]);
        ts.pop_back();
    }
    return ts.front();
}
```

这一算法合并所有的叶子，它在每个迭代都需要扫描列表，因此性能是平方级别的。它可以被进一步提高。观察到每次迭代，只有权重最小的两棵树被合并。为此我们可以使用堆这种数据结构。堆可以保证快速地访问到最小的元素。我们可以将所有的叶子节点放入一个堆中。对于二叉堆，这一个过程需要线性时间。然后我们连续两次从堆顶取出最小元素，将其合并后，再放回堆中。对于二叉堆，这一操作的性能为 $O(\lg n)$。因此，总体性能为 $O(n \lg n)$。这要比上面平方级别的算法要好。下面的算法从堆顶取出元素，然后开始构建 Huffman 树。

$$build(H) = reduce(top(H), pop(H)) \tag{14.89}$$

当堆变空时，算法结束；否则，它从堆顶取出另一棵树进行合并。

$$reduce(T, H) = \begin{cases} T & : & H = \phi \\ build(insert(merge(T, top(H)), pop(H))) & : & otherwise \end{cases} \tag{14.90}$$

函数 *build* 和 *reduce* 互相递归调用。下面的 Haskell 例子程序实现了这一算法。它使用前面章节定义的堆数据结构。

```
huffman' :: (Num a, Ord a) ⇒ [(b, a)] → HTr a b
huffman' = build' ∘ Heap.fromList ∘ map (λ(c, w) → Leaf w c) where
  build' h = reduce (Heap.findMin h) (Heap.deleteMin h)
  reduce x Heap.E = x
  reduce x h = build' $ Heap.insert (Heap.deleteMin h) (merge x (Heap.findMin h))
```

也可以用命令式的方式，使用堆来构造 Huffman 树。首先将全部叶子转换成堆，权重最小的一个置于堆顶。若堆中的元素多于 1 个，我们就取出最小的两个，合并成一棵较大的树，然后放回堆中。重复这一步骤直到堆中剩下最后一棵树，它就是最终的 Huffman 树。

1: **function** HUFFMAN'(A)
2: BUILD-HEAP(A)
3: **while** $|A| > 1$ **do**
4: $T_a \leftarrow$ HEAP-POP(A)
5: $T_b \leftarrow$ HEAP-POP(A)
6: HEAP-PUSH(A, MERGE(T_a, T_b))
7: **return** HEAP-POP(A)

下面的 C++ 例子程序实现了这一使用堆的构建方法。这里使用了标准库中提供的堆。由于缺省情况下是一个最大堆，而非最小堆，因此我们需要传入一个“大于”的比较条件作为参数。

```
bool greaterp(Node* a, Node* b) { return b→w < a→w; }

Node* pop(Nodes& h) {
    Node* m = h.front();
    pop_heap(h.begin(), h.end(), greaterp);
    h.pop_back();
    return m;
}

void push(Node* t, Nodes& h) {
    h.push_back(t);
    push_heap(h.begin(), h.end(), greaterp);
}

Node* huffman1(Nodes ts) {
    make_heap(ts.begin(), ts.end(), greaterp);
    while (ts.size() > 1) {
        Node* t1 = pop(ts);
```

```
        Node* t2 = pop(ts);
        push(merge(t1, t2), ts);
    }
    return ts.front();
}
```

如果字符已经按照权重排序，则存在一个线性时间的构造 Huffman 树的方法。观察 Huffman 树的构造过程，它实际上合并出一系列按照权重递增的树。我们可以用一个队列来管理这些合并好的树。每次我们从队列和树的列表中各取出一棵树，将他们合并起来并放入队列的尾部。处理完列表中的所有树后，队列中将只剩下一棵树。它就是最终的 Huffman 树。在构造过程刚开始的时候，队列为空。

$$build'(A) = reduce'(extract''(\phi, A)) \tag{14.91}$$

这里 A 包含按照权重递增顺序排好序的叶子节点。任何时间，权重最小的树要么在队列的头部，要么是列表中的第一棵树。当队列不空时，记队列头部的树为 T_a，出队后，队列变为 Q'；记 A 中第一棵树为 T_b，剩余的树记为 A'。函数 $extract''$ 可以定义如下。

$$extract''(Q, A) = \begin{cases} (T_b, (Q, A')) & : & Q = \phi \\ (T_a, (Q', A)) & : & A = \phi \vee T_a < T_b \\ (T_b, (Q, A')) & : & otherwise \end{cases} \tag{14.92}$$

实际上，队列和树的列表在整体上可以看作是某种特殊的堆。算法不断将权重最小的树取出然后合并。

$$reduce'(T, (Q, A)) = \begin{cases} T & : & Q = \phi \wedge A = \phi \\ reduce'(extract''(push(Q'', merge(T, T')), A'')) & : & otherwise \end{cases} \tag{14.93}$$

其中 $(T', (Q'', A'')) = extract''(Q, A)$，表示取出另一棵权重最小的树。下面的 Haskell 例子程序实现了这一算法。注意这一程序中，它首先将全部叶子按照权重排序。如果输入的叶子是已序的，就无需这一步。同样，这里使用了列表而并非真正意义上的函数式队列。列表在入队操作时需要线性时间，具体请参考前面关于队列的一章。

```
huffman'' :: (Num a, Ord a) ⇒ [(b, a)] → HTr a b
huffman'' = reduce ∘ wrap ∘ sort ∘ map (λ(c, w) → Leaf w c) where
  wrap xs = delMin ([], xs)
  reduce (x, ([], [])) = x
  reduce (x, h) = let (y, (q, xs)) = delMin h in
                  reduce $ delMin (q ++ [merge x y], xs)
  delMin ([], (x:xs)) = (x, ([], xs))
  delMin ((q:qs), []) = (q, (qs, []))
  delMin ((q:qs), (x:xs)) | q < x = (q, (qs, (x:xs)))
                          | otherwise = (x, ((q:qs), xs))
```

这一算法也可以用命令式的方式实现。

```
1: function HUFFMAN"(A)                                    ▷ A 已按照权重排序
2:     Q ← φ
3:     T ← EXTRACT(Q, A)
4:     while Q ≠ φ ∨ A ≠ φ do
5:         PUSH(Q, MERGE(T, EXTRACT(Q, A)))
6:         T ← EXTRACT(Q, A)
7:     return T
```

其中函数 $\textsc{Extract}(Q, A)$ 从队列和数组中取出权重最小的树。它根据需要会改变队列或者数组。记队列头部的树为 T_a，数组的第一个元素为 T_b。

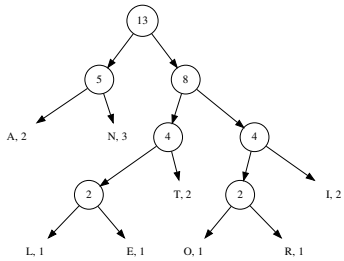
```
1: function EXTRACT(Q, A)
2:     if Q ≠ φ ∧ (A = φ ∨ T_a < T_b) then
3:         return POP(Q)
4:     else
5:         return DETACH(A)
```

其中过程 $\textsc{Detach}(A)$ 将数组 A 的第一个元素取出返回，并从数组中移除。在大多数命令式环境中，从数组中移除第一个元素通常是一个较慢的线性时间操作。我们可以将树按照权重降序存储，这样要移除的就是最后一个元素。速度为常数时间。下面的 C++ 例子程序实现了这一思路。

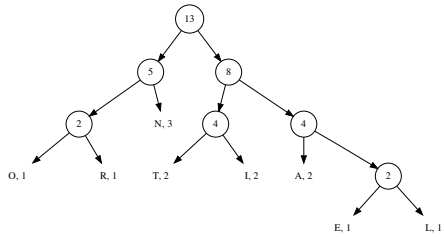
```
Node* extract(queue<Node*>& q, Nodes& ts) {
    Node* t;
    if (!q.empty() && (ts.empty() || lessp(q.front(), ts.back()))) {
        t = q.front();
        q.pop();
    } else {
        t = ts.back();
        ts.pop_back();
    }
    return t;
}

Node* huffman2(Nodes ts) {
    queue<Node*> q;
    sort(ts.begin(), ts.end(), greaterp);
    Node* t = extract(q, ts);
    while (!q.empty() || !ts.empty()) {
        q.push(merge(t, extract(q, ts)));
        t = extract(q, ts);
    }
    return t;
}
```

如果传入的数组是已序的，则无需进行排序。若数组是按照升序传入的，我们可以在线性时间内将其反转。

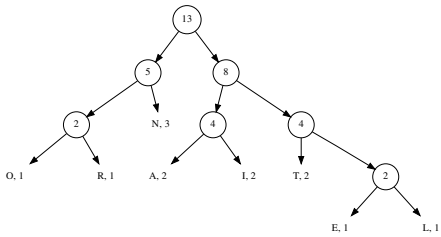
我们介绍了三种 Huffman 树的构造方法。虽然他们都符合 Huffman 提出的策略，但是构造结果却不尽相同。图14.49给出了用三种不同方法构造的 Huffman 树。



(a) 使用扫描方法构造的结果。



(b) 使用堆方法构造的结果。



(c) 对于已序列表，线性时间方法的构造结果。

图 14.49: 同样的字符列表构造出的不同 Huffman 树

虽然这三棵树不同，但是他们都可以产生最高效的编码。这里略过具体的证明，读者可以参考 [91] 或者 [4] 的第 16.3 节了解详细的信息。

Huffman 树的构造过程是 Huffman 编码的核心。可以通过 Huffman 树取得各种结果。例如，通过遍历 Huffman 树可以构造码表。我们用一个空前缀 p，从根节点开始遍历。对于任何分支，如果向左转，我们就在前缀后添加一个 0，如果向右转，就添加一个 1。当到达叶子节点时，就将叶子中的字符和此时的前缀记入码表。记叶子节点中的字符为 c，树 T 的两个分支分别为 T_l 和 T_r。构造码表的函数 $code(T, \phi)$ 定义如下。

$$code(T, p) = \begin{cases} \{(c, p)\} & : & leaf(T) \\ code(T_l, p \cup \{0\}) \cup code(T_r, p \cup \{1\}) & : & otherwise \end{cases} \tag{14.94}$$

其中函数 $leaf(T)$ 检查 T 是一个叶子节点还是分支节点。下面的 Haskell 例子程序根据这一算法产生一个码表的映射。

```
code tr = Map.fromList $ traverse [] tr where
  traverse bits (Leaf _ c) = [(c, bits)]
  traverse bits (Branch _ l r) = (traverse (bits ++ [0]) l) ++
                                 (traverse (bits ++ [1]) r)
```

我们把命令式的码表构造算法留给读者作为练习。编码过程中，我们扫描文本，然后查询码表来输出二进制序列，我们略过其具体的实现。

解码时，我们根据二进制序列查询 Huffman 树。从根节点开始，遇到 0 向左转，遇到 1 向右转。到达叶子节点时，就输出其代表的字符，然后从根节点开始继续解码。当所有二进制序列都消耗完时，解码过程结束。记二进制序列为 $B = \{b_1, b_2, ...\}$，除第一位外的剩余部分为 B'，解码算法可以定义如下。

$$decode(T, B) = \begin{cases} \{c\} & : & B = \phi \wedge leaf(T) \\ \{c\} \cup decode(root(T), B) & : & leaf(T) \\ decode(T_l, B') & : & b_1 = 0 \\ decode(T_r, B') & : & otherwise \end{cases} \tag{14.95}$$

其中 $root(T)$ 返回 Huffman 树的根节点。下面的 Haskell 例子程序实现了解码算法。

```
decode tr cs = find tr cs where
  find (Leaf _ c) [] = [c]
  find (Leaf _ c) bs = c : find tr bs
  find (Branch _ l r) (b:bs) = find (if b == 0 then l else r) bs
```

这是一个在线（on-line）解码算法，性能为线性时间。它每次消耗一个二进制位。这一点可以清楚地从下面的命令式实现中看出，其中的索引每次递增 1。

```
1: function DECODE(T, B)
2:     W ← φ
3:     n ← |B|, i ← 1
4:     while i < n do
5:         R ← T
6:         while ¬ LEAF(R) do
7:             if B[i] = 0 then
8:                 R ← LEFT(R)
9:             else
10:                R ← RIGHT(R)
11:            i ← i + 1
12:        W ← W ∪ SYMBOL(R)
13:    return W
```

下面的 C++ 例子程序实现了这一命令式 Huffman 解码算法。

```
string decode(Node* root, const char* bits) {
    string w;
    while (*bits) {
        Node* t = root;
        while (!isleaf(t))
            t = '0' == *bits++ ? t->left : t->right;
        w += t->c;
    }
    return w;
```

```
}
```

Huffman 编码，特别是 Huffman 树的构造过程展示了一种有趣的策略。每次合并都有若干选项。Huffman 的方法总是从树中选取权重最小的两棵树。这是合并阶段的最好选择。特别地，这一系列局部 最优的选择，产生了一个全局最优的前缀编码。

但并非局部最优选择总能带来全局最优解。在大多数情况下并非如此。Huffman 编码是一个特殊情况。我们称这种每次选择局部最优选项的策略为贪心 策略。

贪心方法可以解决很多问题。但是判断贪心方法能否产生全局最优解却并不容易。通用的形式化证明仍然是一个活跃的研究领域。[4] 中的第 16.4 节介绍了拟阵（Matroid）方法，它覆盖了可以应用贪心算法的很多问题。

换零钱问题

去其他国家前，我们经常要换汇。人们越来越多地使用信用卡了，信用卡很方便，买东西时可以不担心零钱问题。如果使用现金，旅行结束时，往往会剩余一些钱。有些钱币爱好者会把钱换成硬币，收集起来。有没有什么办法，能把指定数量的钱换成最少数量的硬币呢?

我们用美国的钱币系统作为例子。总共有 5 种不同面额的硬币：1 美分、5 美分、25 美分、50 美分、和 1 美元。1 美元等于 100 美分。使用前面介绍的贪心方法，我们每次总挑选不超过余额的最大面值硬币。记硬币价值列表为 $C = \{1, 5, 25, 50, 100\}$。给定任何钱数 X，兑换硬币的方法可以定义如下。

$$change(X, C) = \begin{cases} \phi & : \quad X = 0 \\ \{c_m\} \cup change(X - c_m, C) & : \quad c_m = max(\{c \in C, c \leq X\}) \end{cases} \tag{14.96}$$

如果 C 按照降序排列，c_m 就是第一个不大于 X 的硬币。如果要兑换 1.42 美元，这一函数会生成硬币列表：$\{100, 25, 5, 5, 5, 1, 1\}$。可以很容易地将这一列表变换为一组面值——数量对 $\{(100, 1), (25, 1), (5, 3), (1, 2)\}$。也就是说，我们需要一枚 1 美元硬币、一枚 25 美分硬币、三枚 5 美分硬币、两枚 1 美分硬币。下面的 Haskell 例子程序实现了这一最少兑换算法。

```
solve x = assoc ∘ change x where
  change 0 _ = []
  change x cs = let c = head $ filter (≤ x) cs in c : change (x - c) cs

assoc = (map (λcs → (head cs, length cs))) ∘ group
```

这一程序假设硬币按照降序排列，例如：

```
solve 142 [100, 50, 25, 5, 1]
```

这一算法是尾递归的，他可以很容易地转换为命令式的循环。

1: **function** CHANGE(X, C)
2: $R \leftarrow \phi$

3: **while** $X \neq 0$ **do**
4: $c_m = max(\{c \in C, c \leq X\})$
5: $R \leftarrow \{c_m\} \cup R$
6: $X \leftarrow X - c_m$
7: **return** R

下面的 Python 例子程序实现了这一命令式算法，结果以一个字典输出。

```
def change(x, coins):
    cs = {}
    while x ≠ 0:
        m = max([c for c in coins if c ≤ x])
        cs[m] = 1 + cs.setdefault(m, 0)
        x = x - m
    return cs
```

对于美国这样的硬币系统，贪心方法可以找到最优解。硬币的数量是最少的。幸运的是，贪心算法对于大多数国家的硬币系统都有效。但是也有一些例外。例如，假设某国的硬币体系中包含的币值为 1、3、和 4。如果要兑换价值为 6 的钱，最好的解是使用两个面值为 3 的硬币。但是，贪心方法给出的结果却是 3 枚硬币：一枚面值为 4，两枚面值为 1。这并非最优解。

贪心方法的小结

如换零钱问题所示，贪心方法并不一定能给出最优解。为了找到最优解，我们需要使用后面将要介绍的动态规划方法。

但在实际中，贪心方法得出的解往往还是不错的。举例来说，折行（word-wrap）是现代编辑器、和浏览器等软件中常见的功能。如果文本太长，在一行显示不下，就在某些位置将其拆成若干行显示。使用折行功能，用户就无需在输入时人为加入换行符。虽然动态规划方法能够给出使用最少行的解，但是它过于复杂了。反之，贪心算法能够给出接近最优的折行方案，并且实现起来简单、高效。如下面的算法所示，给定文本 T，每行不能超出宽度 W，每个单词件的间隔为 s。

1: $L \leftarrow W$
2: **for** $w \in T$ **do**
3: **if** $|w| + s > L$ **then**
4: Insert line break
5: $L \leftarrow W - |w|$
6: **else**
7: $L \leftarrow L - |w| - s$

对文本中的每个词 w，该算法使用贪心策略在一行中放入尽可能多的词直到超出行宽限制。很多文本处理软件使用了类似的算法来进行折行处理。

也有很多情况，我们必须要找到严格的最优解，而不是近似最优解。可以使用动

态规划方法来解决此类问题。

动态规划

在介绍换零钱问题时，我们发现贪心方法有时无法得到最优解。对于任何的硬币体系，有没有一种方法，可以保证找到最优解呢?

假设我们找到了兑换价值为 X 的钱的最优方案。所需要的硬币保存在列表 C_m 中。我们可以将这些硬币分成两组，C_1 和 C_2。它们分别等于价值 X_1 和 X_2。我们接下来要证明，C_1 是兑换 X_1 的最优解，且 C_2 是兑换 X_2 的最优解。

证明. 对 X_1，假设存在一个另一个更好的兑换方法 C_1'，它比 C_1 需要的硬币数量更少。则兑换方法 $C_1' \cup C_2$ 使用的硬币数量要少于 C_m。这和 C_m 是兑换价值为 X 的钱的最优解相矛盾。同样，我们也可以证明 C_2 是兑换 X_2 的最优解。 □

注意，相反的情况并不一定成立。如果我们任选一个值 $Y < X$，将原最优兑换问题分解为两个子问题：寻找兑换 Y 的最优解，和寻找兑换 $X - Y$ 的最优解。将这两个最优解合并起来，并不一定是兑换 X 的最优解。考虑这样的反例：有三种硬币，币值为 1、2、和 4。兑换价值为 6 的钱的最优解需要两枚硬币，一枚价值为 2，另一枚价值为 4。但是，如果将问题分解为两个子问题 $6 = 3 + 3$，尽管每个子问题的最优兑换方案为 $3 = 1 + 2$，即使用一枚价值为 1、另一枚价值为 2 的硬币兑换 3，但组合起来的方案需要使用 4 枚硬币 $1 + 2 + 1 + 2$ 来兑换 6。

如果一个最优化问题可以分解为若干子最优化问题，我们称它具备“最优化子结构”(optimal substructure)。兑换零钱问题，必须在硬币价值的基础上分解，而不能任意分解。

兑换零钱问题的最优化子结构可以表达如下。

$$change(X) = \begin{cases} \phi & : \quad X = 0 \\ least(\{c \cup change(X - c) | c \in C, c \leq X\}) & : \quad otherwise \end{cases} \tag{14.97}$$

对于任意硬币系统 C，兑换价值为 0 的钱显然不需要任何硬币；否则，我们检查每一个不大于兑换值 X 的候选币值 c，递归搜索兑换 $X - c$ 的最优解；我们选择所有候选方案中，使用硬币最少的一个作为最终结果。

下面的 Haskell 例子程序实现了这一自顶向下的递归解法。

```
change _ 0 = []
change cs x = minimumBy (compare `on` length)
                [c:change cs (x - c) | c ← cs, c ≤ x]
```

给定输入 `change [1, 2, 4] 6`，即使用价值为 1、2、和 4 的硬币，兑换价值为 6 的钱，这一程序可以给出正确的答案 `[2, 4]`。尽管如此，它在解决使用美国硬币体系兑换 1.42 美元的问题时性能成为了瓶颈。在一台 2.7GHz 的 CPU，拥有 8G 内存的计算机上，这一程序在 15 分钟内仍未得出结果。

造成性能问题的原因在于，在自顶向下递归求解中，有大量的重复计算。当计算 $change(142)$ 时，它需要检查 $change(141)$、$change(137)$、$change(117)$、$change(92)$、和 $change(42)$。接着在计算 $change(141)$ 时，它需要将这个值分别减去 1、2、25、50、和 100 美分。这样，就会再次遇到 137、117、92、和 42 这些值。搜索空间按照 5 的指数急速爆炸。

这和使用自顶向下的递归方法计算斐波那契序列非常相似。

$$F_n = \begin{cases} 1 & : \quad n = 1 \vee n = 2 \\ F_{n-1} + F_{n-2} & : \quad otherwise \end{cases} \tag{14.98}$$

举例来说，当计算 F_8 的时候，我们需要递归计算 F_7 和 F_6。而当在计算 F_7 时，我们需要再次计算 F_6，以及 F_5……展开的过程如下面的等式，每次展开，计算都加倍。相同的值被一遍一遍地重复计算。

$$\begin{aligned} F_8 = & \quad F_7 + F_6 \\ = & \quad F_6 + F_5 + F_5 + F_4 \\ = & \quad F_5 + F_4 + F_4 + F_3 + F_4 + F_3 + F_3 + F_2 \\ = & \quad ... \end{aligned}$$

为了避免重复计算，我们可以在求斐波那契数的时候维护一个表格 F。这个表格的前两个元素被填写为 1，其他的元素都是空白。在自顶向下的递归计算中，如果需要计算 F_k，我们首先检查表格中的第 k 个元素，如果不是空白，我们就直接使用表格中的值。否则，我们需要进一步计算。当计算出 F_k 的值后，我们将其保存入表格中，以用于后继的查找。

```
1: F ← {1, 1, NIL, NIL, ...}
2: function FIBONACCI(n)
3:     if n > 2 ∧ F[n] = NIL then
4:         F[n] ← FIBONACCI(n − 1) + FIBONACCI(n − 2)
5:     return F[n]
```

使用类似的思路，我们可以得出一个新的自顶向下的兑换硬币方法。我们使用一个表格 T 来记录最优的兑换办法。开始的时候，所有的内容都为空白。在自顶向下的递归计算中，我们查询这个表格，寻找兑换较小价值钱的兑换方法。每当计算出新值的兑换方法后，我们都把它存入表格中。

```
1: T ← {φ, φ, ...}
2: function CHANGE(X)
3:     if X > 0 ∧ T[X] = φ then
4:         for c ∈ C do
5:             if c ≤ X then
6:                 C_m ← {c} ∪ CHANGE(X − c)
7:                 if T[X] = φ ∨ |C_m| < |T[X]| then
```

8: $T[X] \leftarrow C_m$

9: **return** $T[X]$

兑换价值为 0 的钱，显然不需要任何硬币，所以解为空 ϕ。否则，我们查找 $T[X]$ 获得兑换 X 的解。如果表格中这项为空，则需要递归计算。我们在 C 中逐一尝试所有币值不大于 X 的硬币，寻找子问题，即兑换价值为 $X - c$ 的最优方法。在子问题的最优解基础上，我们再加上 1 枚价值为 c 的硬币，就获得了兑换 X 的最优解。然后，我们将此最优解保存在表格中的 $T[X]$ 一项内。

下面的 Python 例子程序实现了这一算法，它仅使用 8000 毫秒就给出了兑换 1.42 美元的最优解。

```
tab = [[] for _ in range(1000)]

def change(x, cs):
    if x > 0 and tab[x] == []:
        for s in [[c] + change(x - c, cs) for c in cs if c <= x]:
            if tab[x] == [] or len(s) < len(tab[x]):
                tab[x] = s
    return tab[x]
```

另外一种计算斐波那契数的方法，是按照顺序 $F_1, F_2, F_3, ..., F_n$ 来计算。这恰好是人们在依次写下斐波那契数时的顺序。

1: **function** FIBO(n)

2: $F = \{1, 1, NIL, NIL, ...\}$

3: **for** $i \leftarrow 3$ to n **do**

4: $F[i] \leftarrow F[i-1] + F[i-2]$

5: **return** $F[n]$

我们可以使用类似的思路来解决兑换硬币问题。从价值为 0 的钱开始，所需硬币为空，然后我们接着寻找如何兑换价值为 1 的钱。以美国硬币系统为例，我们可以使用 1 美分；接着对于价值为 2、3、和 4 的钱，可以分别兑换为 2 枚 1 美分硬币、3 枚 1 美分硬币、和 4 枚 1 美分硬币。此时保存最优解的列表内容如表14.12(a) 所示。

价值	0	1	2	3	4
最优解	ϕ	$\{1\}$	$\{1,1\}$	$\{1,1,1\}$	$\{1,1,1,1\}$

(a) 兑换价值为 4 美分以内的最优解列表

价值	0	1	2	3	4	5
最优解	ϕ	$\{1\}$	$\{1,1\}$	$\{1,1,1\}$	$\{1,1,1,1\}$	$\{5\}$

(b) 兑换价值为 5 美分以内的最优解列表

表 14.12: 兑换零钱的最优解列表

当兑换价值为 5 的钱时，情况发生了变化。共有两个选择：再次使用一枚 1 美

分的硬币，即使用 5 个 1 美分的硬币兑换；或者使用 1 枚 5 美分的硬币。显然后者所需的硬币更少。因此最优解表格的内容变为如表14.12(b) 所示。

接下来，兑换价值为 6 的钱时，由于有两种硬币：1 美分和 5 美分都不大于 6，我们需要检查这 2 种选项。

- 如果选择使用 1 美分，我们接下来需要兑换剩余的价值 5。由于我们已经在表格中记录了兑换 5 的最优解 $\{5\}$，使用 1 枚 5 美分硬币。这样我们就得到一个兑换价值为 6 的一个解 $\{5, 1\}$；
- 如果选择使用 5 美分，我们接下来需要兑换剩余的价值 1。通过查表，我们发现兑换 1 的最优解 $\{1\}$，这样我们就得到了兑换价值为 6 的另外一个解 $\{1, 5\}$。

恰巧两个选项获得解都只需要两枚硬币，我们可以选择任何一个作为最优解。原则上说，我们每次选择所需硬币最少的解填入表格中。

在任何一次迭代中，当寻找价值 $i < X$ 的兑换方案时，我们逐一检查所有的币值。对于任何不大于 i 的硬币，我们从表格中查找项 $T[i-c]$ 来获取子问题的解。用这一解所需的硬币再加上一枚硬币 c，就是兑换 i 的一个方案选项。我们选择所需硬币最少的一个，记录到表格中。

下面的算法实现了这一自底向上的思路。

1: **function** CHANGE(X)
2: $T \leftarrow \{\phi, \phi, ...\}$
3: **for** $i \leftarrow 1$ to X **do**
4: **for** $c \in C, c \leq i$ **do**
5: **if** $T[i] = \phi \vee 1 + |T[i-c]| < |T[i]|$ **then**
6: $T[i] \leftarrow \{c\} \cup T[i-c]$
7: **return** $T[X]$

下面的 Python 例子程序实现了这一算法。

```
def changemk(x, cs):
    s = [[] for _ in range(x+1)]
    for i in range(1, x+1):
        for c in cs:
            if c <= i and (s[i] == [] or 1 + len(s[i-c]) < len(s[i])):
                s[i] = [c] + s[i-c]
    return s[x]
```

观察保存解的表格，会发现其中有大量重复的内容。

价值	6	7	8	9	10	...
最优解	$\{1, 5\}$	$\{1, 1, 5\}$	$\{1, 1, 1, 5\}$	$\{1, 1, 1, 1, 5\}$	$\{5, 5\}$	...

表 14.13: 最优解的表格中存在重复内容

这是因为最优子问题的解，被完全复制到父问题的解中。为了减少空间的消耗，我们可以仅记录相对子问题变化的部分。对于兑换硬币问题，我们只需要记录下为了兑换 i，所选择的那一枚硬币。

```
1: function CHANGE'(X)
2:     T ← {0, ∞, ∞, ...}
3:     S ← {NIL, NIL, ...}
4:     for i ← 1 to X do
5:         for c ∈ C, c ≤ i do
6:             if 1 + T[i − c] < T[i] then
7:                 T[i] ← 1 + T[i − c]
8:                 S[i] ← c
9:     while X > 0 do
10:        PRINT(S[X])
11:        X ← X − S[X]
```

为了避免记录完整的兑换硬币列表，这一新算法使用了两个表格 T 和 S。T 记录了兑换价值 0、1、2……所需的最少硬币数量，而 S 记录了最优解所选择的第一个币值。为了获得兑换 X 的完整硬币列表，第一个选择的硬币为 $S[X]$，接下来的最优化子问题是兑换 $X' = X - S[X]$。我们查询表格 $S[X']$ 获得下一个硬币。我们不断查询最优化子问题所需选择的硬币，直到表格的最初位置。下面的 Python 例子程序实现了这一算法。

```
def chgmk(x, cs):
    cnt = [0] + [x+1] * x
    s = [0]
    for i in range(1, x+1):
        coin = 0
        for c in cs:
            if c ≤ i and 1 + cnt[i-c] < cnt[i]:
                cnt[i] = 1 + cnt[i-c]
                coin = c
        s.append(coin)
    r = []
    while x > 0:
        r.append(s[x])
        x = x - s[x]
    return r
```

给定需要兑换的值 n，这一算法循环 n 次。每次迭代，算法最多检查全部的硬币。总体运行时间为 $\Theta(nk)$，其中 k 是指定硬币系统中不同面值硬币的数量。最后改进的算法，需要额外 $O(n)$ 的空间。它使用表格 T 和 S 来记录最优化子问题的解。

在纯函数式的环境中，我们无法更改记录解的表格，或者在常数时间内查询。一种办法是使用前面章节介绍的 finger 树[12]。我们可以把所需的最少硬币数，和选择的

[12] 某些纯函数式编程环境，如 Haskell，提供了内置的数组；而其他的一些近似纯函数式环境，如 ML，提供了可改

硬币成对保存在树中。

记录最优解的表格，实际上为一棵 finger 树，它初始为 $T = \{(0, 0)\}$。表示兑换价值为 0 的钱，无需任何硬币。我们对列表 $\{1, 2, ..., X\}$ 进行 fold，传入初始表格。fold 使用的二元函数是 $change(T, i)$。fold 结束后，我们获得最终的最优解表格，然后再通过函数 $make(X, T)$，从这一表格构造出兑换硬币的列表。

$$makeChange(X) = make(X, fold(change, \{(0, 0)\}, \{1, 2, ..., X\})) \tag{14.99}$$

在函数 $change(T, i)$ 中，我们检查所有价值不大于 i 的硬币，选出导致最优解的一个。所需硬币的最少数量，和选中的硬币组成一对值，插入到 finger 树中，最后返回新的表格作为结果。

$$change(T, i) = insert(T, fold(sel, (\infty, 0), \{c | c \in C, c \leq i\})) \tag{14.100}$$

我们再次使用 fold 来选择硬币数最少的兑换方案。fold 起始时的值为 $(\infty, 0)$，列表为所有面值不大于 i 的硬币。函数 $sel((n, c), c')$ 接受两个参数，第一个参数是一对值，包含所需硬币数量和选中的硬币。它是目前为止找到的最优解；另一个参数是一枚新硬币，我们需要检查这枚新硬币是否可以导致更好的解。

$$sel((n, c), c') = \begin{cases} (1 + n', c') & : \quad 1 + n' < n, (n', c') = T[i - c'] \\ (n, c) & : \quad otherwise \end{cases} \tag{14.101}$$

构造好最优解表格后，兑换所需的所有硬币就可以通过它逐一找出。

$$make(X, T) = \begin{cases} \phi & : \quad X = 0 \\ \{c\} \cup make(X - c, T) & : \quad (n, c) = T[X] \end{cases} \tag{14.102}$$

下面的 Haskell 例子程序实现了兑换硬币算法。它使用了标准库中的 `Data.Sequence`，其实现为 finger 树。

```
import Data.Sequence (Seq, singleton, index, (|>))

changemk x cs = makeChange x $ foldl change (singleton (0, 0)) [1..x] where
  change tab i = let sel c = min (1 + fst (index tab (i - c)), c)
                 in tab |> (foldr sel ((x + 1), 0) $ filter (≤i) cs)
  makeChange 0 _ = []
  makeChange x tab = let c = snd $ index tab x in c : makeChange (x - c) tab
```

不管是自底向上的方法，还是自顶向下的方法，都需要记录最优化子问题的解。这是因为在计算整体的最优解时，需要反复多次使用子问题的结果。这一特性称为重叠子问题（overlapping sub problems）。

变的数组。

动态规划的性质

动态规划最早在 1940 年代由 Richard Bellman 提出。它是搜索最优解的有利武器，它要求问题要具备两个性质。

- 最优化子结构。问题可以被分解为若干规模较小的子问题，最优解可以高效地从这些子问题的解中构造出来；
- 重叠子问题。问题可以被分解为若干子问题，子问题的解被多次反复使用以寻找整体上的解。

兑换硬币问题，同时拥有最优化子结构和重叠子问题的性质。

最长公共子序列问题

最长公共子序列问题和最长公共子串问题不同。在后缀树一章中，我们给出了如何寻找最长公共子串的方法。最长公共子序列无需是原序列中的连续部分。

例如，文本“Mississippi”和“Missunderstanding”的最长公共子串为“Miss”，而最长公共子序列为“Misssi”。如图14.50所示。

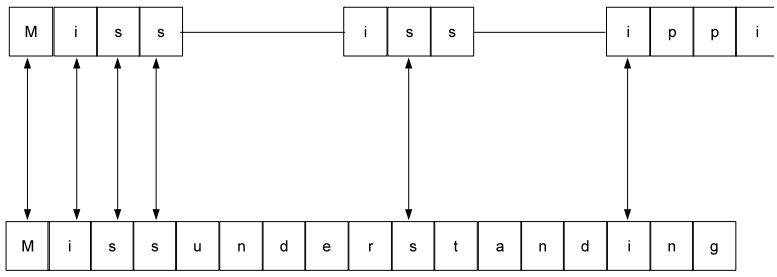


图 14.50: 最长公共子序列

如果我们将这张图旋转 90 度，然后考虑这两段文本代表两段代码，它就变成了代码间比较“diff”的结果。大多数现在版本控制工具需要计算不同版本间的差异。最长公共子序列问题在其中扮演了重要的角色。

如果两个字符串 X 和 Y 中的任何一个为空，则最长公共子序列 $LCS(X, Y)$ 也显然为空。否则，记 $X = \{x_1, x_2, ..., x_n\}$、$Y = \{y_1, y_2, ..., y_m\}$。如果第一个元素 x_1 和 y_1 相同，我们可以递归地搜索 $X' = \{x_2, x_3, ..., x_n\}$ 和 $Y' = \{y_2, y_3, ..., y_m\}$ 的最

长公共子序列。最终结果 $LCS(X, Y)$ 可以通过将 x_1 附加到 $LCS(X', Y')$ 之前获得。否则，若 $x_1 \neq y_1$，我们需要递归搜索 $LCS(X, Y')$ 和 $LCS(X', Y)$ 的结果，选择较长的一个作为最终结果。综合这三种情况，我们可以得到下面的定义。

$$LCS(X, Y) = \begin{cases} \phi & : \quad X = \phi \vee Y = \phi \\ \{x_1\} \cup LCS(X', Y') & : \quad x_1 = y_1 \\ longer(LCS(X, Y'), LCS(X', Y)) & : \quad otherwise \end{cases} \tag{14.103}$$

这一定义中含有明显的最优化子结构，最长公共子序列问题可以分解为规模较小的子问题。子问题至少比原问题的字符串长度短 1。

同样，这一定义中也含有重叠子问题。子串间的最长公共子序列被多次用于搜索全局最优解。

由于存在这两个性质，我们可以使用动态规划来解决这一问题。

我们可以使用一个二维表格来记录子问题的最优解。行和列分别代表 X 和 Y 的子串。

		a	n	t	e	n	n	a
		1	2	3	4	5	6	7
b	1							
a	2							
n	3							
a	4							
n	5							
a	6							

表 14.14: 记录最优解的二维表格

这一表格给出了求字符串“antenna”和“banana”之间最长公共子序列的例子。两个字符串的长度分别为 7 和 6. 我们首先检查表格的右下角，由于这一项为空，我们需要比较“antenna”中的第 7 个字符，和“banana”中的第 6 个字符。它们都是字符‘a’，我们接下来要递归查找第 5 行、第 6 列。这一项也为空，我们需要重复这一过程，直到达到边界情况，即一个字符串变为空，或者我们查找的表格中的一项已填入信息。同兑换硬币问题类似，当某一子问题的最优解被找到后，它被记录到表格中用于后继的查找。这一过程和递归定义相比，顺序是相反的，我们从每个字符串中最右侧的字符开始处理。

考虑空串和任何字符串的最长公共子序列也为空，我们可以扩展上述表格，使得第一行和第一列包含空字符串。

下面的算法通过使用这样的表格，实现了自顶向下的动态规划解法。

```
1: T ← NIL
2: function LCS(X, Y)
```

		a	n	t	e	n	n	a
		ϕ	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ
b	ϕ							
a	ϕ							
n	ϕ							
a	ϕ							
n	ϕ							
a	ϕ							

表 14.15: 包含空串的最优解表格

3: $m \leftarrow |X|, n \leftarrow |Y|$
4: $m' \leftarrow m + 1, n' \leftarrow n + 1$
5: **if** $T = \text{NIL}$ **then**
6: $T \leftarrow \{\{\phi, \phi, ..., \phi\}, \{\phi, NIL, NIL, ...\}, ...\}$ ▷ $m' \times n'$
7: **if** $X \neq \phi \wedge Y \neq \phi \wedge T[m'][n'] = \text{NIL}$ **then**
8: **if** $X[m] = Y[n]$ **then**
9: $T[m'][n'] \leftarrow \text{APPEND}(\text{LCS}(X[1...m-1], Y[1...n-1]), X[m])$
10: **else**
11: $T[m'][n'] \leftarrow \text{LONGER}(\text{LCS}(X, Y[1...n-1]), \text{LCS}(X[1...m-1], Y))$
12: **return** $T[m'][n']$

表格初始化时，第一行和第一列都被填入空串；剩余的项都为 NIL。除非任何一个字符串为空，或者表格中的项不为 NIL，我们比较两个字符串中的最后一个元素，并且递归计算子串间的最长公共子序列。下面的 Python 例子程序实现了这一算法。

```
def lcs(xs, ys):
    m = len(xs)
    n = len(ys)
    global tab
    if tab is None:
        tab = [[""]*(n+1)] + [[""] + [None]*n for _ in xrange(m)]
    if m ≠ 0 and n ≠0 and tab[m][n] is None:
        if xs[-1] == ys[-1]:
            tab[m][n] = lcs(xs[:-1], ys[:-1]) + xs[-1]
        else:
            (a, b) = (lcs(xs, ys[:-1]), lcs(xs[:-1], ys))
            tab[m][n] = a if len(b) < len(a) else b
    return tab[m][n]
```

也可以用自底向上的方法寻找最长公共子序列。思路和兑换硬币问题类似。另外，我们还可以避免在表格中保存完整的序列内容，而只存储最长子序列的长度，并最终从表格中构造出最长公共子序列。一开始时，表格中的所有项都初始化为 0。

```
1: function LCS(X, Y)
2:     m ← |X|, n ← |Y|
3:     T ← {{0, 0, ...}, {0, 0, ...}, ...}                    ▷ (m + 1) × (n + 1)
4:     for i ← 1 to m do
5:         for j ← 1 to n do
6:             if X[i] = Y[j] then
7:                 T[i + 1][j + 1] ← T[i][j] + 1
8:             else
9:                 T[i + 1][j + 1] ← MAX(T[i][j + 1], T[i + 1][j])
10:    return GET(T, X, Y, m, n)

11: function GET(T, X, Y, i, j)
12:    if i = 0 ∨ j = 0 then
13:        return φ
14:    else if X[i] = Y[j] then
15:        return APPEND(GET(T, X, Y, i − 1, j − 1), X[i])
16:    else if T[i − 1][j] > T[i][j − 1] then
17:        return GET(T, X, Y, i − 1, j)
18:    else
19:        return GET(T, X, Y, i, j − 1)
```

自底向上的搜索时，我们从第 2 行、第 2 列开始。这一项对应 X 和 Y 中的第 1 个元素。如果它们相等，则目前为止的最长公共子序列的长度为 1。这可以通过将空串的长度加 1 得到。而空串的结果，存储在左上角上。否则，如果第一个元素不等，我们从表格正上方的一项和左方的一项中挑选较大的值填入。重复这一步骤，最终填完表格。

此后，我们进行回溯以构造出最长公共子序列。从表格的右下方开始。如果 X 和 Y 中最后一个元素相等，我们就把它作为结果中的最后一个元素，并沿着对角线方向继续查表。否则，如果最后一个元素不等，我们需要比较左侧和上方的项，选择值较大的继续进行处理。

下面的 Python 例子程序实现了这一算法。

```
def lcs(xs, ys):
    m = len(xs)
    n = len(ys)
    c = [[0]*(n+1) for _ in xrange(m+1)]
    for i in xrange(1, m+1):
        for j in xrange(1, n+1):
            if xs[i-1] == ys[j-1]:
                c[i][j] = c[i-1][j-1] + 1
            else:
                c[i][j] = max(c[i-1][j], c[i][j-1])
```

```
    return get(c, xs, ys, m, n)

def get(c, xs, ys, i, j):
    if i==0 or j==0:
        return []
    elif xs[i-1] == ys[j-1]:
        return get(c, xs, ys, i-1, j-1) + [xs[i-1]]
    elif c[i-1][j] > c[i][j-1]:
        return get(c, xs, ys, i-1, j)
    else:
        return get(c, xs, ys, i, j-1)
```

也可以用纯函数式的方法定义自底向上的动态规划解法。我们还是用 finger 树作为表格。第一行填入 $n+1$ 个 0。通过对序列 X 进行 fold 来构造表格。然后再从表格中构造最长公共子序列。

$$LCS(X, Y) = construct(fold(f, \{\{0, 0, ..., 0\}\}, zip(\{1, 2, ...\}, X))) \tag{14.104}$$

由于需要按照索引查表，我们将 X 和自然数 zip 到一起。函数 f 通过对 Y 进行 fold，创建表格中新的一行，并记录下目前为止，所有情况下最长公共子序列的长度。

$$f(T, (i, x)) = insert(T, fold(longest, \{0\}, zip(\{1, 2, ...\}, Y))) \tag{14.105}$$

函数 $longest$ 接受两个参数，第一个参数是目前为止表格中这一行已填入的内容，第二个参数是一对值，包含一个索引和 Y 中对应的元素。它比较这一元素和 X 中是否相同，并将较长的长度添加到这一行中。

$$longest(R, (j, y)) = \begin{cases} insert(R, 1 + T[i-1][j-1]) & : & x = y \\ insert(R, max(T[i-1][j], T[i][j-1])) & : & otherwise \end{cases} \tag{14.106}$$

表格构建完成后，可以通过查表构造出最长公共子序列。为了提高效率，我们可以传入反转的序列 $\overleftarrow{X}$ 和 $\overleftarrow{Y}$，以及他们各自的长度 m 和 n。

$$construct(T) = get((\overleftarrow{X}, m), (\overleftarrow{Y}, n)) \tag{14.107}$$

如果序列不为空，记两个序列中的第一个元素分别为 x 和 y。剩余的部分记为 $\overleftarrow{X}'$ 和 $\overleftarrow{Y}'$。函数 get 的具体定义如下。

$$get((\overleftarrow{X}, i), (\overleftarrow{Y}, j)) = \begin{cases} \phi & : & \overleftarrow{X} = \phi \wedge \overleftarrow{Y} = \phi \\ get((\overleftarrow{X}', i-1), (\overleftarrow{Y}', j-1)) \cup \{x\} & : & x = y \\ get((\overleftarrow{X}', i-1), (\overleftarrow{Y}, j)) & : & T[i-1][j] > T[i][j-1] \\ get((\overleftarrow{X}, i), (\overleftarrow{Y}', j-1)) & : & otherwise \end{cases} \tag{14.108}$$

下面的 Haskell 例子程序实现了这一算法。

```
lcs' xs ys = construct $ foldl f (singleton $ fromList $ replicate (n+1) 0)
                                 (zip [1..] xs) where
  (m, n) = (length xs, length ys)
  f tab (i, x) = tab |> (foldl longer (singleton 0) (zip [1..] ys)) where
    longer r (j, y) = r |> if x == y
                      then 1 + (tab `index` (i-1) `index` (j-1))
                      else max (tab `index` (i-1) `index` j) (r `index` (j-1))
  construct tab = get (reverse xs, m) (reverse ys, n) where
    get ([], 0) ([], 0) = []
    get ((x:xs), i) ((y:ys), j)
      | x == y = get (xs, i-1) (ys, j-1) ++ [x]
      | (tab `index` (i-1) `index` j) > (tab `index` i `index` (j-1)) =
                get (xs, i-1) ((y:ys), j)
      | otherwise = get ((x:xs), i) (ys, j-1)
```

子集和问题

动态规划不仅可以解决最优化问题，还可以解决一些更为一般的搜索问题。例如子集和（subset sum）问题。给定若干整数的集合，是否存在一个非空子集，使得子集中元素相加的结果为 0? 例如集合 $\{11, 64, -82, -68, 86, 55, -88, -21, 51\}$ 存在两个和为 0 的非空子集。一个是 $\{64, -82, 55, -88, 51\}$，另一个是 $\{64, -82, -68, 86\}$。

当然 0 是一个特殊的情况，有时我们需要找到一个子集，使得其和为某一给定值 s。本节中，我们要找到所有满足的子集。

显然，暴力穷举法可以找到所有的解。对于每个元素，我们可以选择或者排除它，因此对于有 n 个元素的集合，总共有 2^n 个选项。对于每个选项，我们都需要检查和是否为 s。累加是一个线性操作。因此总体上的复杂度为 $O(n2^n)$。这是一个指数级的算法，如果集合中含有很多元素，所需时间会急速增加。

子集和问题存在一个递归解。如果集合为空，显然无解；否则，令集合为 $X = \{x_1, x_2, ...\}$。若 $x_1 = s$，则子集 $\{x_1\}$ 是一个解，我们接着需要搜索集合的剩余部分 $X' = \{x_2, x_3, ...\}$ 中是否仍有子集的和为 s。否则，若 $x_1 \neq s$，则存在两种可能性。我们既需要在 X' 中搜索子集和 s，也需要搜索子集和 $s - x_1$。对于任何和为 $s - x_1$ 的子集，我们可以将 x_1 加入集合，构成一个新的解。下面的定义总结了上述的所有情况。

$$solve(X, s) = \begin{cases} \phi & : \quad X = \phi \\ \{\{x_1\}\} \cup solve(X', s) & : \quad x_1 = s \\ solve(X', s) \cup \{\{x_1\} \cup S | S \in solve(X', s - x_1)\} & : \quad otherwise \end{cases} \tag{14.109}$$

这一定义明显含有子结构，虽然它不是最优化子结构。并且，这一定义也含有重叠子问题。我们可以用动态规划的思路，使用一张表来记录子问题的解，从而解决子集和问题。

在输出所有满足的子集内容前，我们首先考虑如何解决判定问题。当存在某一子集和为 s，则输出“存在”，否则输出“不存在”。

通过一轮扫描我们可以确定子集和的上下限。如果指定的和 s 不在上下限确定的范围内，则显然无解。

$$\begin{cases} s_l = \sum\{x \in X, x < 0\} \\ s_u = \sum\{x \in X, x > 0\} \end{cases} \tag{14.110}$$

否则，若 $s_l \leq s \leq s_u$，由于所有的元素都是整数，我们可以使用一张表格，含有 $s_u - s_l + 1$ 列，每列代表这一范围内的一个可能的值，从 s_l 到 s_u。表格中每项的内容为真或假，表示是否存在一个子集，其和为该项对应的值。开始的时候，所有的项都初始化为假。我们从集合 X 中的第一个元素 x_1 开始，显然子集 $\{x_1\}$ 的和为 x_1，所以表格第一行中代表 x_1 的一项应为真。

	s_l	s_l+1	...	x_1	...	s_u
x_1	F	F	...	T	...	F

表 14.16: 子集和问题的解表格第一行

使用集合中的第二个元素 x_2，可以得到 3 种可能的和。和第一行类似，子集 $\{x_2\}$ 的和为 x_2；对于前一行中所有可能值，如果不加上 x_2，它们作为子集和仍然可以得到，所以第一行中 x_1 下面的一项也应该为真；通过将 x_2 加到所有可能的和之上，我们可以得到一些新值。因此代表 $x_1 + x_2$ 的一项应为真。

	s_l	s_l+1	...	x_1	...	x_2	...	x_1+x_2	...	s_u
x_1	F	F	...	T	...	F	...	F	...	F
x_2	F	F	...	T	...	T	...	T	...	F

表 14.17: 子集和问题的解表格第二行

总而言之，当填写表格第 i 行的时候，所有由 $\{x_1, x_2, ..., x_{i-1}\}$ 可获得的和，仍然可以获得。因此上一行中为真的项，仍然为真。对应值为 x_i 的一项应为真，因为只含有一个元素的集合 $\{x_i\}$ 的和为 x_i。我们可以将 x_i 加到已知的所有和之上，这样可以得到一些新值，对应这些新值的项也应为真。

当这样处理完所有的元素后，我们得到了一个含有 $|X|$ 行的表格。通过查询最后一行，对应值为 s 的项是真还是假，就可以知道是否存在子集的和为 s。由于 $s < s_l$ 或 $s_u < s$ 时无解，在这种情况下可以跳过表格的构造过程。我们暂时略过这一错误处理。

```
1: function SUBSET-SUM(X, s)
2:     s_l ← ∑{x ∈ X, x < 0}
3:     s_u ← ∑{x ∈ X, x > 0}
```

```
    n ← |X|
    T ← {{False, False, ...}, {False, False, ...}, ...}        ▷ n × (s_u − s_l + 1)
    for i ← 1 to n do
        for j ← s_l to s_u do
            if X[i] = j then
                T[i][j] ← True
            if i > 1 then
                T[i][j] ← T[i][j] ∨ T[i − 1][j]
                j' ← j − X[i]
                if s_l ≤ j' ≤ s_u then
                    T[i][j] ← T[i][j] ∨ T[i − 1][j']
    return T[n][s]
```

注意，表格中列的索引不是从 1 到 $s_u - s_l + 1$，而是从 s_l 到 s_u。由于大多数编程环境不支持负索引，我们可以通过 $T[i][j - s_l]$ 来进行换算。下面的 Python 例子程序使用了负索引的特性。

```
def solve(xs, s):
    low = sum([x for x in xs if x < 0])
    up  = sum([x for x in xs if x > 0])
    tab = [[False]*(up-low+1) for _ in xs]
    for i in xrange(0, len(xs)):
        for j in xrange(low, up+1):
            tab[i][j] = (xs[i] == j)
            j1 = j - xs[i];
            tab[i][j] = tab[i][j] or tab[i-1][j] or
                        (low ≤ j1 and j1 ≤ up and tab[i-1][j1])
    return tab[-1][s]
```

这一程序没有使用单独的分支来处理 $i = 0$ 和 $i = 1, 2, ..., n - 1$ 的不同情况。这是因为 $i = 0$ 时，行的索引 $i - 1 = -1$，它指向表格中的最后一行，其中的值都为假。这样就简化了程序的逻辑。

使用这一表格，可以很容易地构建出所有和为 s 的子集。首先查询表格中最后一行代表 s 的一项。如果最后一个元素 $x_n = s$，则子集 $\{x_n\}$ 显然是一个解。我们接下来查找上一行中 s 对应的项，并递归地从 $\{x_1, x_2, x_3, ..., x_{n-1}\}$ 中构造所有和为 s 的子集。最后，我们检查倒数第二行，对应 $s - x_n$ 的项。对于所有和为这一值的子集，我们加入 x_n 后构造一个新的集合，其和为 s。

```
function GET(X, s, T, n)
    S ← φ
    if X[n] = s then
        S ← S ∪ {X[n]}
    if n > 1 then
```

```
6:        if T[n − 1][s] then
7:            S ← S ∪ GET(X, s, T, n − 1)
8:        if T[n − 1][s − X[n]] then
9:            S ← S ∪ {{X[n]} ∪ S′|S′ ∈ GET(X, s − X[n], T, n − 1) }
10:   return S
```

下面的 Python 例子程序实现了这一算法。

```
def get(xs, s, tab, n):
    r = []
    if xs[n] == s:
        r.append([xs[n]])
    if n > 0:
        if tab[n-1][s]:
            r = r + get(xs, s, tab, n-1)
        if tab[n-1][s - xs[n]]:
            r = r + [[xs[n]] + ys for ys in get(xs, s - xs[n], tab, n-1)]
    return r
```

这一子集和问题的动态规划解法循环了 $O(n(s_u - s_l + 1))$ 次以构建表格，然后递归 $O(n)$ 次从这一表格构造最后的解。它所用的空间也为 $O(n(s_u - s_l + 1))$。

我们可以使用一个向量来代替含有 n 行的表格。向量中的每一项对应一个可能的和，其中存储了子集的列表。开始时，向量中的所有项都为空。对于 X 中的每一个元素，我们不断更新向量，它记录了所有可能得到的和。当所有的元素都处理完毕后，对应 s 的那一项包含了最终的答案。

```
1:  function SUBSET-SUM(X, s)
2:      s_l ← Σ{x ∈ X, x < 0}
3:      s_u ← Σ{x ∈ X, x > 0}
4:      T ← {φ, φ, ...}                                  ▷ s_u − s_l + 1
5:      for x ∈ X do
6:          T′ ← DUPLICATE(T)
7:          for j ← s_l to s_u do
8:              j′ ← j − x
9:              if x = j then
10:                 T′[j] ← T′[j] ∪ {x}
11:             if s_l ≤ j′ ≤ s_u ∧ T[j′] ≠ φ then
12:                 T′[j] ← T′[j] ∪ {{x} ∪ S|S ∈ T[j′]}
13:         T ← T′
14:     return T[s]
```

下面的 Python 例子程序实现了这一改进的算法。

```
def subsetsum(xs, s):
    low = sum([x for x in xs if x < 0])
```

```
    up  = sum([x for x in xs if x > 0])
    tab = [[] for _ in xrange(low, up+1)]
    for x in xs:
        tab1 = tab[:]
        for j in xrange(low, up+1):
            if x == j:
                tab1[j].append([x])
            j1 = j - x
            if low ≤ j1 and j1 ≤ up and tab[j1] ≠ []:
                tab1[j] = tab1[j] + [[x] + ys for ys in tab[j1]]
        tab = tab1
    return tab[s]
```

这一命令式算法含有一个清晰的结构，通过逐一处理每个元素，最终构造出保存解的表格。这可以通过 fold 以纯函数式的方式实现。我们仍然使用手指树作为向量，从 s_l 伸展到 s_u。最开始时所有项均为空。

$$subsetsum(X, s) = fold(build, \{\phi, \phi, ..., \}, X)[s] \tag{14.111}$$

经过 fold 后，解表格就构造好了，通过查询第 s 项就可以得到最终的答案[13]。

对于每一元素 $x \in X$，函数 $build$ 对列表 $\{s_l, s_l + 1, ..., s_u\}$ 进行 fold，对于每个值 j，检查它是否等于 x，并且将只有 1 个元素的集合 $\{x\}$ 加入第 j 项。注意这里索引从 s_l 开始，而不是从 0 开始。如果对应值 $j - x$ 的项不为空，则复制其中的所有子集，并将元素 x 加入到每个集合中。

$$build(T, x) = fold(f, T, \{s_l, s_l + 1, ..., s_u\}) \tag{14.112}$$

$$f(T, j) = \begin{cases} T'[j] \cup \{\{x\} \cup Y | Y \in T[j']\} & : \quad s_l \le j' \le s_u \wedge T[j'] \neq \phi, j' = j - x \\ T' & : \quad otherwise \end{cases} \tag{14.113}$$

这里函数 f 对 T' 进行调整，而 T' 的定义如下：

$$T' = \begin{cases} \{x\} \cup T[j] & : \quad x = j \\ T & : \quad otherwise \end{cases} \tag{14.114}$$

式 (14.113) 和 (14.114) 的第一行都返回一个新表格，其中的某些项根据给定值进行了更新。

下面的 Haskell 例子程序实现了这一算法。

```
subsetsum xs s = foldl build (fromList [[] | _ ← [l..u]]) xs `idx` s where
  l = sum $ filter (< 0) xs
  u = sum $ filter (> 0) xs
  idx t i = index t (i - l)
  build tab x = foldl (λt j → let j' = j - x in
                  adjustIf (l ≤ j' && j' ≤ u && tab `idx` j' ≠ [])
                           (++ [(x:ys) | ys ← tab `idx` j']) j
                           (adjustIf (x == j) ([x]:) j t)) tab [l..u]
  adjustIf pred f i seq = if pred then adjust f (i - l) seq else seq
```

[13]这里，我们再次跳过了 $s < s_l$ 或 $s > s_u$ 的错误处理，如果 s 不在上下限范围内，则无解。

一些材料，如 [72] 针对动态规划抽象出了一些公共结构。为了解决具体的问题，只需要向通用解中提供特定的前置条件，定义好如何决定一个解优于另一个，以及如何将子问题的解合并。但是在实际中，问题往往多种多样，十分复杂。我们需要仔细地分析问题的各种性质。

练习 14.3

- 使用栈来找出迷宫问题的所有解。
- 八皇后问题存在 92 个不同的解。对于任何一个解，将其旋转 90°、180°、270° 也都是八皇后问题的解。并且在水平和垂直方向翻转也能产生解。有些解是对称的，因此旋转或者翻转后的解是同一个。在这个意义上说，真正不同的解只有 12 个。修改八皇后的程序，找出这 12 个不同的解。改进程序，使用较少的搜索步骤找出 92 个解。
- 改进八皇后的算法，使得它可以解决 n 皇后问题。
- 修改跳跃青蛙问题的函数式解法，使得它可以解决每侧 n 只青蛙的情况。
- 修改狼、羊、白菜问题的算法，找出所有可能的解。
- 给出完整的扩展欧几里得算法以解决倒水问题。
- 我们无需知道具体的线性组合系数 x 和 y。通过最大公约数得知问题可解后，我们可以机械地执行这样的过程：倒满 A，将 A 中的水倒入 B，当 B 满后，将其倒空。直到某一个瓶中得到了指定容积的水。请实现这一解法。它是否能比最初的解法更快找到解？
- 和扩展欧几里得方法相比，广度优先搜索法可以说是某种意义上的暴力搜索。改进扩展欧几里得算法，寻找最好的线性组合使得 $|x| + |y|$ 最小。
- 康威（John Horton Conway）提出了一种滑动趣题。图14.51给出的是一种简化的版本。8 个圆圈中的 7 个已经放入了棋子，每个棋子上标有编号 1 到 7。如果和棋子相邻的圆圈是空的，则棋子可以滑动过去。圆圈间如果有连线，则表示它们是相连的。目标是将棋子从顺序 1、2、3、4、5、6、7 通过滑动反转成 7、6、5、4、3、2、1。编写一个程序解决康威滑动问题。
- 实现命令式的 Huffman 码表生成算法。
- 对最长公共子序列问题，另一种自底向上的解法是子表格中记录“方向”，而不是序列的长度。有三个值：‘N’ 代表向北，‘W’ 代表向西，‘NW’ 代表向西北。这些方向指示我们如何构建最终的结果。我们从表格的右下角开始，如果值为 ‘NW’，我们就沿着对角线移动到左上方的格子；如果值为 ‘N’，就垂直移动到上方的格子；如果为 ‘W’，就水平移动到左侧的格子。选择一门编程语言，实现这一算法。

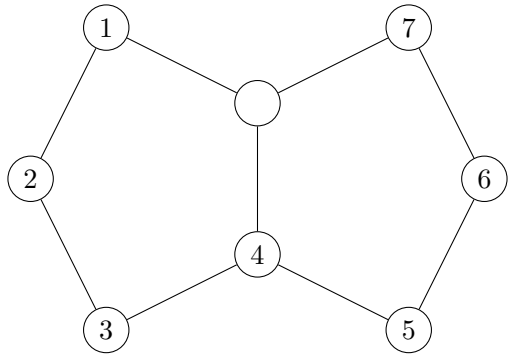


图 14.51: 康威滑动趣题

- Levenshtein 编辑距离是一种衡量两个字符串相似程度的量。它定义为从字符串 s 转换到字符串 t 所需花费的成本。它被广泛用于拼写检查，OCR 纠错等场景中。Levenshtein 编辑距离允许三种操作：增加一个字符、删除一个字符、替换一个字符。每种操作每次只改变一个字符，下面的例子中，给出了如何从字符串“kitten”转换到“sitting”的过程，从而得出其 Levenshtein 编辑距离为 3。

  1. $\underline{\text{k}}\text{itten} \rightarrow \underline{\text{s}}\text{itten}$ （将 k 替换为 s）；
  2. $\text{sitt}\underline{\text{e}}\text{n} \rightarrow \text{sitt}\underline{\text{i}}\text{n}$ （将 e 替换为 i）；
  3. $\text{sitten} \rightarrow \text{sitting}\underline{\ }$ （在结尾处插入 g）。

  使用动态规划，计算两个字符串间的 Levenshtein 距离。

14.4 小结

本章介绍了基本的搜索方法。有些方法通过扫描在数据中寻找感兴趣的信息，它们通常具有某些结构，可以在扫描中不断更新已知的信息。这可以看作是信息重用的某种特殊情况。Boyer-Moore 众数问题、最大子序列和问题、以及字符串匹配算法都是这一类方法的例子。另一种常用的搜索策略是分而治之，通过不断减小搜索域的规模，直到找出期望的结果。典型的 k 选择问题、二分查找、以及 Saddleback 搜索都应用了分而治之的策略。本章还介绍了一些搜索问题解的方法，这些解往往不是待搜索的特定元素，它们可以是一系列的决策，或者是某种有组织的操作。深度优先和广度优先搜索法是最简单的两类解搜索策略。如果一个问题存在多个解，有时人们希望寻找最优解，动态规划方法被广泛用来解决含有最优子结构的问题。对于某些特殊情况，我们还可以使用简化的策略，例如贪心策略，以较小的代价获得最优解。

附录 A　红黑树的命令式删除算法

本附录包含红黑树的命令式删除算法。我们需要在普通二叉搜索树删除算法的基础上，通过旋转和重新染色恢复红黑树的性质，以保持树的平衡。我们在红黑树一章中指出，当删除黑色节点时，会破坏红黑树的第五条性质。使得某一路径上的黑色节点数目减少。为此，我们引入“双重黑色”节点，来保持所删除路径上的黑色节点数目不变。

A.1　双重黑色

为了支持“双重黑色”的节点，我们需要增加颜色的定义。如下面的 C++ 例子代码所示。

```
enum class Color { RED, BLACK, DOUBLY_BLACK };
```

在删除一个节点时，我们复用二叉搜索树的删除算法，并记录被删除节点的父节点。如果被删除节点的颜色是黑色我们需要通过处理保持黑色的属性，然后再进行进一步修复。

```
1:  function DELETE(T, x)
2:      p ← PARENT(x)
3:      q ← NIL
4:      if LEFT(x) = NIL then
5:          q ← RIGHT(x)
6:          replace x with RIGHT(x)
7:      else if RIGHT(x) = NIL then
8:          q ← LEFT(x)
9:          replace x with LEFT(x)
10:     else
11:         y ← MIN(RIGHT(x))
12:         p ← PARENT(y)
13:         q ← RIGHT(y)
14:         KEY(x) ← KEY(y)
15:         copy satellite data from y to x
```

```
16:         replace y with RIGHT(y)
17:         x ← y
18:     if COLOR(x) = BLACK then
19:         T ← DELETE-FIX(T, MAKE-BLACK(p, q), q = NIL?)
20:     release x
21:     return T
```

删除算法接受树的根节点 T 和待删除节点 x。如果待删除节点存在一个为空的分支，我们可以将 x “切下”，并用另一个分支 q 来替代 x。否则，我们在 x 的右子树中找到最小的节点 y，用 y 替换 x。然后递归地将 y “切下”。如果被删除的节点 x 的颜色为黑色，我们调用 MAKE-BLACK(p, q)，来保持黑色属性，以便进行下一步的修复。

```
1: function MAKE-BLACK(p, q)
2:     if p = NIL ∧ q = NIL then
3:         return NIL                          ▷ 删除只有一个叶子节点的树后，变为空
4:     else if q = NIL then
5:         n ← Doubly Black NIL
6:         PARENT(n) ← p
7:         return n
8:     else
9:         return BLACKEN(q)
```

如果传入 MAKE-BLACK 的参数 p 和 q 都为空，说明我们在删除只有一个叶子节点的树，删除后树变为空。否则如果父节点 p 不为空，而节点 q 为空。说明我们删除了一个黑色的叶子节点。这相当于，此时一个 NIL 节点替换了被删除的黑色节点。根据红黑树的性质 3，NIL 节点实际上都是黑色的。我们可以把这一 NIL 节点变成“双重黑色”NIL 节点来保持其所在路径上的黑色节点数目不变。最后，如果 p、q 都不为空，我们调用 BLACKEN 检查 q 的颜色，如果是红色的，将它重新染成黑色，如果 q 已经是黑色的，我们将它染成双重黑色。

A.1.1 修复

为了最终恢复红黑树的性质，我们需要通过树的旋转操作和重新染色，最终去掉“双重黑色”。这里有三种情况需要处理（[4] 第 292 页）。每一种情况中，双重黑色的节点即可以是普通节点，也可以是双重黑色的空节点。我们首先看第一种情况。

A.1.2 双重黑色节点的兄弟为黑色，并且该兄弟节点有一个红色子节点

对于这种情况，我们可以通过旋转操作来修复。总共有四种不同的细分情况，它们全部可以变换到一种统一的形式。如图A.1所示。

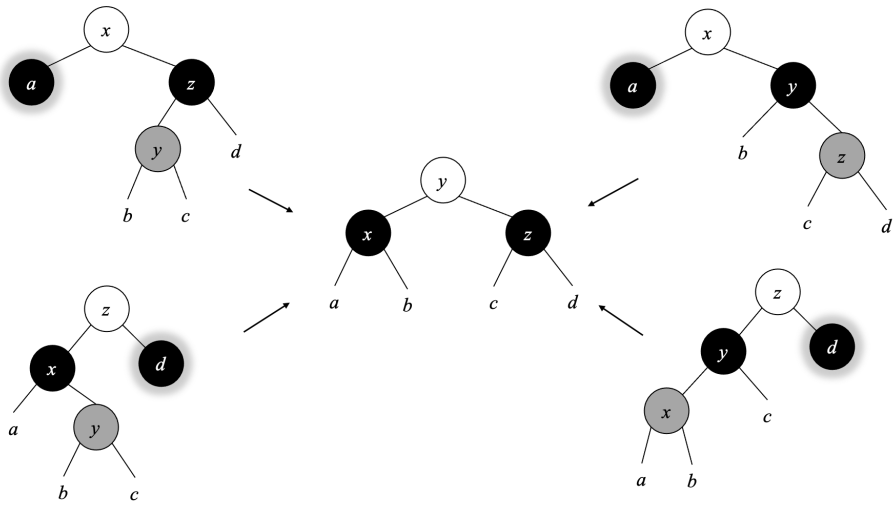


图 A.1: 双重黑色节点的兄弟为黑色，并且该兄弟节点有一个红色子节点。这种情况可以通过一次旋转操作来修复。

下面的算法描述了针对这一情况的处理。

```
1:  function DELETE-FIX(T, x, f)
2:      n ← NIL
3:      if f = True then                          ▷ x 是一个双重黑色 NIL 节点
4:          n ← x
5:      if x = NIL then                           ▷ 将只有一个叶子节点的树删空
6:          return NIL
7:      while x ≠ T∧ COLOR(x) = B² do             ▷ x 不是根且 x 是双重黑色
8:          if SIBLING(x) ≠ NIL then              ▷ 双重黑色节点的兄弟节点不为空
9:              s ← SIBLING(x)
10:             ...
11:             if s is black ∧ LEFT(s) is red then      ▷ 兄弟为黑，一个侄子为红
12:                 if x = LEFT(PARENT(x)) then          ▷ x 在左侧
13:                     set x, PARENT(x), and LEFT(s) all black
14:                     T ← ROTATE-RIGHT(T, s)
15:                     T ← ROTATE-LEFT(T, PARENT(x))
16:                 else                                 ▷ x 在右侧
17:                     set x, PARENT(x), s, and LEFT(s) all black
18:                     T ← ROTATE-RIGHT(T, PARENT(x))
19:             else if s is black ∧ RIGHT(s) is red then ▷ 兄弟为黑，一个侄子为红
20:                 if x = LEFT(PARENT(x)) then          ▷ x 在左侧
21:                     set x, PARENT(x), s, and RIGHT(s) all black
```

```
22:             T ← RB-Left-Rotate(T, Parent(x))
23:         else                                          ▷ x 在右侧
24:             set x, Parent(x), and Right(s) all black
25:             T ← RB-Left-Rotate(T, s)
26:             T ← RB-Right-Rotate(T, Parent(x))
27:         ...
```

A.1.3 双重黑色节点的兄弟节点为红色

这种情况下，我们可以通过旋转，将双重黑色恢复为普通的黑色节点。如图A.2所示，将左侧的树旋转到右侧的结构后，可将双重黑色的节点 a 或 c 恢复为黑色。

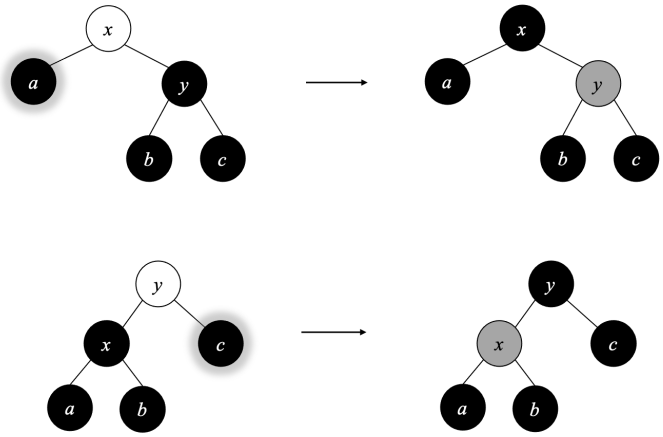


图 A.2: 双重黑色节点的兄弟节点为红色

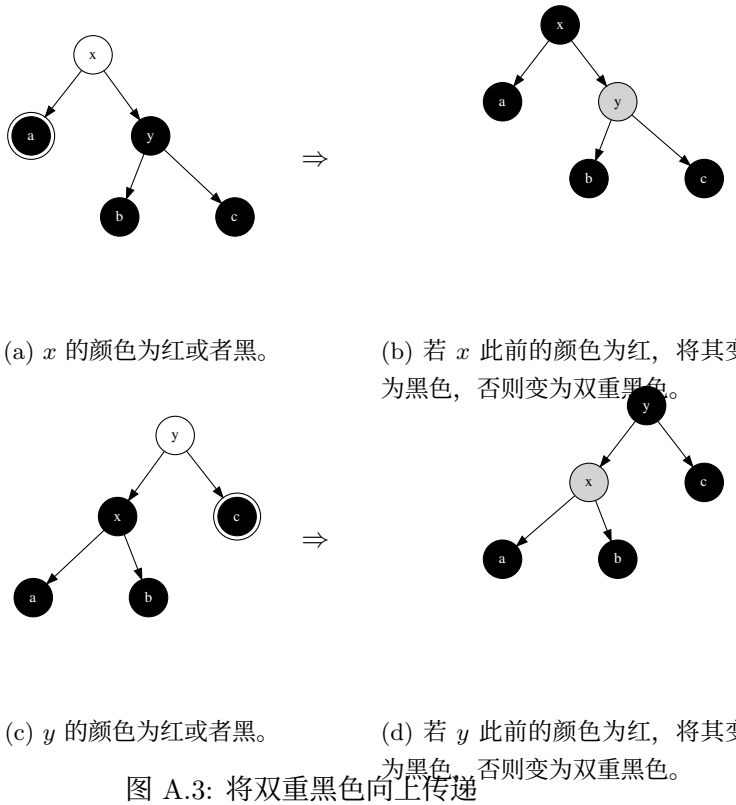
我们在此前给出算法上增加这一处理。

```
1: function Delete-Fix(T, x, f)
2:     n ← NIL
3:     if f = True then                          ▷ x 是一个双重黑色 NIL 节点
4:         n ← x
5:     if x = NIL then                           ▷ 将只有一个叶子节点的树删空
6:         return NIL
7:     while x ≠ T ∧ Color(x) = B² do            ▷ x 不是根且 x 是双重黑色
8:         if Sibling(x) ≠ NIL then              ▷ 双重黑色节点的兄弟节点不为空
9:             s ← Sibling(x)
10:            if s is red then                  ▷ 兄弟为红色
11:                set Parent(x) red
12:                set s black
13:                if x = Left(Parent(x)) then   ▷ x 在左侧
14:                    T ← RB-Left-RotateT, Parent(x)
```

```
15:            else                                          ▷ x 在右侧
16:                T ← RotateRightT, Parent(x)
17:        else if s is black ∧ Left(s) is red then    ▷ 兄弟为黑，一个侄子为红
18:            ...
```

A.1.4 双重黑色节点的兄弟节点为黑色，该兄弟节点的两个子节点也全是黑色。

这种情况下，我们可以将这个兄弟节点染成红色，将双重黑色变回黑色，然后将双重黑色属性向上传递一层到父节点。如图A.3所示，有两种对称的情况。



(a) x 的颜色为红或者黑。

(b) 若 x 此前的颜色为红，将其变为黑色，否则变为双重黑色。

(c) y 的颜色为红或者黑。

(d) 若 y 此前的颜色为红，将其变为黑色，否则变为双重黑色。

图 A.3: 将双重黑色向上传递

上述三种情况中，双重黑色节点的兄弟节点都不为空。如果其兄弟节点为空，我们可以直接将双重黑色恢复为普通黑色，然后将黑色向上传递。如果双重黑色最终向上传递到根节点，我们可以将根节点变为普通黑色节点，从而结束修复过程。另外，如果双重黑色在修复过程中被重新染色为普通节点，我们也可以终止。最后，我们需要一个额外处理，即如果最终的双重黑色节点是一个双重黑色空节点，我们需要将其恢复为普通空节点。最终的算法如下所示。

```
1: function Delete-Fix(T, x, f)
2:     n ← NIL
3:     if f = True then                                  ▷ x 是一个双重黑色 NIL 节点
```

```
4:      n ← x
5:  if x = NIL then                                  ▷ 将只有一个叶子节点的树删空
6:      return NIL
7:  while x ≠ T∧ Color(x) = B² do                    ▷ x 不是根且 x 是双重黑色
8:      if Sibling(x) ≠ NIL then                     ▷ 双重黑色节点的兄弟节点不为空
9:          s ← Sibling(x)
10:         if s is red then                         ▷ 兄弟为红色
11:             set Parent(x) red
12:             set s black
13:             if x = Left(Parent(x)) then          ▷ x 在左侧
14:                 T ← Rotate-LeftT, Parent(x)
15:             else                                 ▷ x 在右侧
16:                 T ← Rotate-RightT, Parent(x)
17:         else if s is black ∧ Left(s) is red then ▷ 兄弟为黑，一个侄子为红
18:             if x = Left(Parent(x)) then          ▷ x 在左侧
19:                 set x, Parent(x), and Left(s) all black
20:                 T ← Rotate-Right(T, s)
21:                 T ← Rotate-Left(T, Parent(x))
22:             else                                 ▷ x 在右侧
23:                 set x, Parent(x), s, and Left(s) all black
24:                 T ← Rotate-Right(T, Parent(x))
25:         else if s is black ∧ Right(s) is red then ▷ 兄弟为黑，一个侄子为红
26:             if x = Left(Parent(x)) then          ▷ x 在左侧
27:                 set x, Parent(x), s, and Right(s) all black
28:                 T ← Rotate-Left(T, Parent(x))
29:             else                                 ▷ x 在右侧
30:                 set x, Parent(x), and Right(s) all black
31:                 T ← Rotate-Left(T, s)
32:                 T ← Rotate-Right(T, Parent(x))
33:         else if s, Left(s), and Right(s) are all black then ▷ 兄弟和侄子都为黑色
34:             set x black
35:             set s red
36:             Blacken(Parent(x))
37:             x ← Parent(x)
38:     else                                         ▷ 无兄弟节点，将黑色向上传递
39:         set x black
```

```
            BLACKEN(PARENT(x))
            x ← PARENT(x)
    set T black
    if n ≠ NIL then                    ▷ 替换双重黑色的 NIL 节点为普通 NIL
        replace n with NIL
    return T
```

删除修复时，传入了三个参数，一个是树的根节点 T；一个是待修复的节点 x，它可能是双重黑色的节点；还有一个标记 f，如果带修复的节点 x 是双重黑色的空节点 NIL，则 f 为真。此时我们用 n 来记录这一双重黑色的空节点 NIL，并且在最终修复完毕后，用普通 NIL 替换掉 n。

下面的 C++ 例子程序实现了红黑树的删除算法。

```
Node* del(Node* t, Node* x) {
    if (!x) return t;
    Node* parent = x->parent;
    Node* db = nullptr;        //doubly black
    Node* y;

    if (x->left == nullptr) {
        db = x->right;
        x->replaceWith(db);
    } else if (x->right == nullptr) {
        db = x->left;
        x->replaceWith(db);
    } else {
        y = min(x->right);
        parent = y->parent;
        db = y->right;
        x->key = y->key;
        y->replaceWith(db);
        x = y;
    }
    if (x->color == Color::BLACK)
        t = deleteFix(t, makeBlack(parent, db), db == nullptr);
    remove(x);
    return t;
}
```

其中 `makeBlack` 函数检查删除后节点是否变为双重黑色，并处理双重黑色空节点 NIL 的特殊情况。

```
Node* makeBlack(Node* parent, Node* x) {
    if (!parent && ! x)
        return nullptr;
    if (!x)
        return Node::replace(parent, x, new Node(0, Color::DOUBLY_BLACK));
    return blacken(x);
}
```

其中函数 replace(parent, x, y) 将 parent 的子节点 x，用 y 替换。

```
static Node* replace(Node* parent, Node* x, Node* y) {
    if (parent == nullptr) {
        if (y) y->parent = nullptr;
    } else if (parent->left == x) {
        parent->setLeft(y);
    } else {
        parent->setRight(y);
    }
    if (x) x->parent = nullptr;
    return y;
}
```

函数 blacken(node) 将红色节点染为黑色，将黑色节点染为双重黑色。

```
Node* blacken(Node* x) {
    x->color = isRed(x) ? Color::BLACK : Color::DOUBLY_BLACK;
    return x;
}
```

最终的修复过程的实现如下。

```
Node* deleteFix(Node* t, Node* db, bool isDBEmpty) {
    Node* dbEmpty = isDBEmpty ? db : nullptr;
    if (!db) return nullptr;    // remove the root from a leaf tree;
    while (db ≠ t && db->color == Color::DOUBLY_BLACK) {
        if (db->sibling() ≠ nullptr) {
            if (isRed(db->sibling())) {
                // the sibling is red, (transform to make the sibling black)
                setColors(db->parent, Color::RED,
                          db->sibling(), Color::BLACK);
                if (db == db->parent->left)
                    t = leftRotate(t, db->parent);
                else
                    t = rightRotate(t, db->parent);
            } else if (isBlack(db->sibling()) && isRed(db->sibling()->left)) {
                // the sibling is black, and one nephew is red
                if (db == db->parent->left) {
                    setColors(db, Color::BLACK,
                              db->parent, Color::BLACK,
                              db->sibling()->left, db->parent->color);
                    t = rightRotate(t, db->sibling());
                    t = leftRotate(t, db->parent);
                } else {
                    setColors(db, Color::BLACK,
                              db->parent, Color::BLACK,
                              db->sibling(), db->parent->color,
                              db->sibling()->left, Color::BLACK);
                    t = rightRotate(t, db->parent);
                }
            } else if (isBlack(db->sibling()) && isRed(db->sibling()->right)) {
                if (db == db->parent->left) {
                    setColors(db, Color::BLACK,
```

```
                            db→parent, Color::BLACK,
                            db→sibling(), db→parent→color,
                            db→sibling()→right, Color::BLACK);
                    t = leftRotate(t, db→parent);
                } else {
                    setColors(db, Color::BLACK,
                            db→parent, Color::BLACK,
                            db→sibling()→right, db→parent→color);
                    t = leftRotate(t, db→sibling());
                    t = rightRotate(t, db→parent);
                }
            } else if (isBlack(db→sibling()) &&
                       isBlack(db→sibling()→left) &&
                       isBlack(db→sibling()→right)) {
                // the sibling and both nephews are black.
                // move the blackness up
                setColors(db, Color::BLACK,
                          db→sibling(), Color::RED);
                blacken(db→parent);
                db = db→parent;
            }
        } else { // no sibling, move the blackness up
            db→color = Color::BLACK;
            blacken(db→parent);
            db = db→parent;
        }
    }
    t→color = Color::BLACK;
    if (dbEmpty) { // change the doubly black nil to nil
        dbEmpty→replaceWith(nullptr);
        delete dbEmpty;
    }
    return t;
}
```

其中 `isBlack(node)` 判断一个节点是否为黑色，根据红黑树的性质，所有的 NIL 节点的颜色为黑色。

```
bool isBlack(Node* x) {
    return x == nullptr || x→color == Color::BLACK;
}

bool isRed(Node* x) {
    return x ≠ nullptr && x→color == Color::RED;
}
```

函数 `setColors` 为一组辅助函数，用于对节点染色。

```
void setColors(Node* x, Color a, Node* y, Color b) { x→color = a; y→color = b; }

void setColors(Node* x, Color a, Node* y, Color b, Node* z, Color c) {
    setColors(x, a, y, b);
    z→color = c;
```

```
}

void setColors(Node* x, Color a, Node* y, Color b,
               Node* z, Color c, Node* q, Color d) {
    setColors(x, a, y, b);
    setColors(z, c, q, d);
}
```

算法在结束前修复双重黑色空节点 NIL 时，调用节点的 `replaceWith` 方法，它通过调用前面定义的 `replace` 函数实现。

```
void replaceWith(Node* y) {
    replace(parent, this, y);
}
```

考虑红黑树的平衡性，删除算法或者到达根节点终止，或者双重黑色消失终止。对于含有 n 个节点的红黑树，删除算法的复杂度为 $O(\lg n)$。

练习 A.1

- 选择一种命令式编程语言，实现完整的红黑树删除算法。
- 选择一种命令式编程语言，编写程序判断一个给定的红黑树是否满足红黑树的 5 条性质，并用这一程序测试红黑树的删除算法。

附录 B　AVL 树——证明和删除算法

本附录首先给出 AVL 树模式匹配插入算法的证明。

B.1　插入后树高度的变化

向 AVL 树插入一个 key 后，高度的增加存在 4 种情况。

$$\begin{aligned}\Delta H &= |T'| - |T| \\ &= \begin{cases} \Delta H_r & : \Delta \geq 0 \wedge \Delta' \geq 0 \\ \Delta + \Delta H_r & : \Delta \leq 0 \wedge \Delta' \geq 0 \\ \Delta H_l - \Delta & : \Delta \geq 0 \wedge \Delta' \leq 0 \\ \Delta H_l & : otherwise \end{cases}\end{aligned}$$

为了证明这一结论，首先注意到一次插入操作不可能同时增加左右分支的高度，因此我们可以做上述分解。根据定义，平衡因子等于右子树的高度减去左子树的高度。这 4 种情况可以分别解释如下：

- 如果 $\Delta \geq 0$ 并且 $\Delta' \geq 0$。这说明在插入前后，右子树的高度都不小于左子树的高度。因子整个树高度的增加，全部“贡献”自右子树高度的变化 ΔH_r；
- 如果 $\Delta \leq 0$，说明在插入前，左子树的高度不小于右子树。但是插入后 $\Delta' \geq 0$，说明右子树的高度由于插入操作增加了，而左子树的高度保持不变（$|T_l'| = |T_l|$）。所以高度的增加为：

$$\begin{aligned}\Delta H &= max(|T_r'|, |T_l'|) - max(|T_r|, |T_l|) && \{\Delta \leq 0 \wedge \Delta' \geq 0\} \\ &= |T_r'| - |T_l| && \{|T_l| = |T_l'|\} \\ &= |T_r| + \Delta H_r - |T_l| \\ &= \Delta + \Delta H_r\end{aligned}$$

- 如果 $\Delta \geq 0$ 且 $\Delta' \leq 0$，和第二种情况类似，我们有：

$$\begin{aligned}\Delta H &= max(|T_r'|, |T_l'|) - max(|T_r|, |T_l|) && \{\Delta \geq 0 \wedge \Delta' \leq 0\} \\ &= |T_l'| - |T_r| \\ &= |T_l| + \Delta H_l - |T_r| \\ &= \Delta H_l - \Delta\end{aligned}$$

- 最后一种情况，Δ 和 Δ' 都不大于 0，说明插入前后左子树的高度都不小于右子树。所以高度的增加全部“贡献”自左子树的变化 ΔH_l。

B.2 插入后平衡调整算法的证明

如图5.3所示，所有需要修复平衡的 4 种情况中，平衡因子都是－ 2 或 2。调整后，平衡因子恢复为 0。左右子树具有相同的高度。

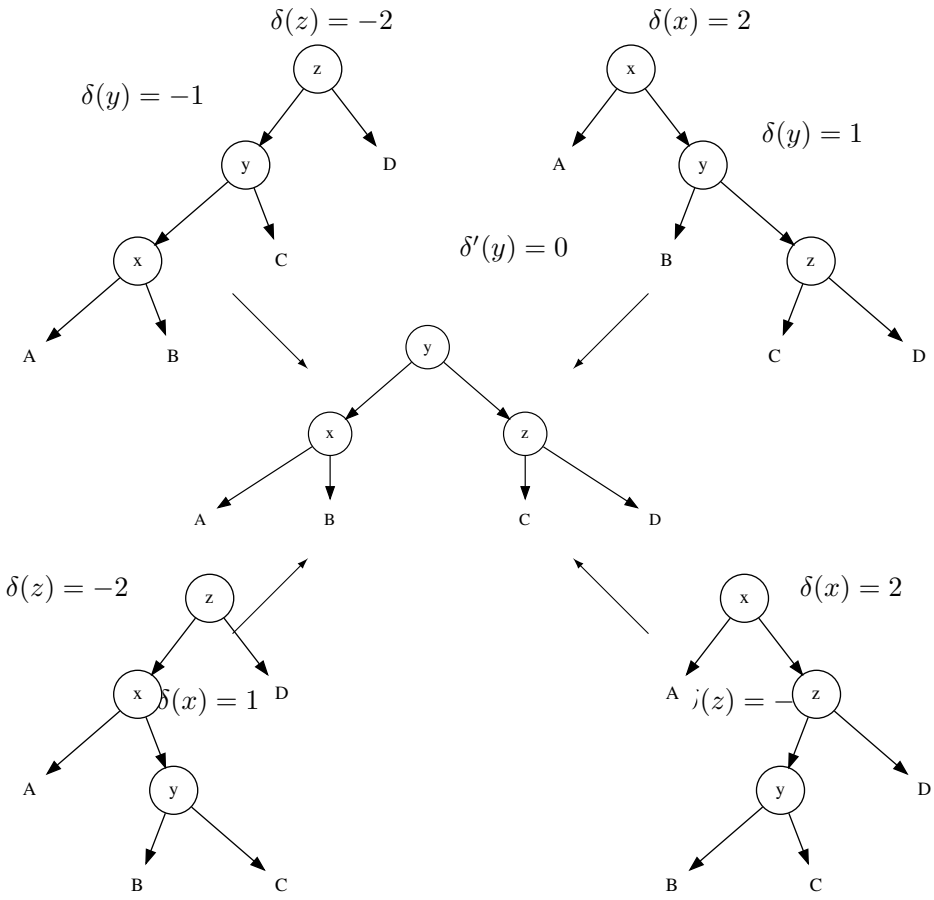


图 B.1: 插入后需要恢复平衡的 4 种情况

这 4 种情况分别是：左－左偏、右－右偏、右－左偏、左－右偏。记修复前的平衡因子分别为 $\delta(x)$、$\delta(y)$、$\delta(z)$，修复后的平衡因子分别为 $\delta'(x)$、$\delta'(y)$、$\delta'(z)$。

我们接下来将证明，经过调整后，所有 4 种情况的平衡因子都变成 $\delta(y) = 0$。并且将给出调整后 $\delta'(x)$ 和 $\delta'(z)$ 的结果。

左－左偏（Left-left lean）的情况

由于 x 子分支在调整前后的结构维持不变，因此可以立即得到等式：$\delta'(x) = \delta(x)$。因为 $\delta(y) = -1$ 且 $\delta(z) = -2$，所以：

$$\begin{aligned} \delta(y) &= |C| - |x| = -1 \Rightarrow |C| = |x| - 1 \\ \delta(z) &= |D| - |y| = -2 \Rightarrow |D| = |y| - 2 \end{aligned} \tag{B.1}$$

调整平衡后：

$$\begin{aligned} \delta'(z) &= |D| - |C| && \{\text{根据式(B.1)}\} \\ &= |y| - 2 - (|x| - 1) \\ &= |y| - |x| - 1 && \{x\text{是}y\text{的子节点} \Rightarrow |y| - |x| = 1\} \\ &= 0 \end{aligned} \tag{B.2}$$

对于 $\delta'(y)$，调整平衡后我们有如下结果：

$$\begin{aligned} \delta'(y) &= |z| - |x| \\ &= 1 + max(|C|, |D|) - |x| && \{\text{根据式 (B.2)，我们有}|C| = |D|\} \\ &= 1 + |C| - |x| && \{\text{根据式 (B.1)}\} \\ &= 1 + |x| - 1 - |x| \\ &= 0 \end{aligned} \tag{B.3}$$

汇总上述结果，对于左－左偏的情况，新的平衡因子如下：

$$\begin{aligned} \delta'(x) &= \delta(x) \\ \delta'(y) &= 0 \\ \delta'(z) &= 0 \end{aligned} \tag{B.4}$$

右－右偏（Right-right lean）的情况

因为右－右偏和左－左偏对称，易知新的平衡因子结果如下：

$$\begin{aligned} \delta'(x) &= 0 \\ \delta'(y) &= 0 \\ \delta'(z) &= \delta(z) \end{aligned} \tag{B.5}$$

右－左偏（Right-left lean）的情况

首先考虑 $\delta'(x)$。调整平衡后，我们有：

$$\delta'(x) = |B| - |A| \tag{B.6}$$

调整平衡前，如果我们计算 z 的高度，有如下的结果：

$$\begin{aligned} |z| &= 1 + max(|y|, |D|) \quad \{\delta(z) = -1 \Rightarrow |y| > |D|\} \\ &= 1 + |y| \\ &= 2 + max(|B|, |C|) \end{aligned} \tag{B.7}$$

因为 $\delta(x) = 2$，所以可以推出：

$$\begin{aligned} \delta(x) = 2 &\Rightarrow |z| - |A| = 2 \quad \{\text{根据式 (B.7)}\} \\ &\Rightarrow 2 + max(|B|, |C|) - |A| = 2 \\ &\Rightarrow max(|B|, |C|) - |A| = 0 \end{aligned} \tag{B.8}$$

如果 $\delta(y) = 1$，也就是 $|C| - |B| = 1$，则有下面的关系：

$$max(|B|, |C|) = |C| = |B| + 1 \tag{B.9}$$

将其代入式 (B.8) 得到：

$$\begin{aligned} &|B| + 1 - |A| = 0 \Rightarrow |B| - |A| = -1 \quad \{\text{根据式 (B.6) }\} \\ &\Rightarrow \delta'(x) = -1 \end{aligned} \tag{B.10}$$

反之，如果 $\delta(y) \neq 1$，则有 $max(|B|, |C|) = |B|$，将其代入式 (B.8) 得到：

$$\begin{aligned} &|B| - |A| = 0 \quad \{\text{根据式 (B.6)}\} \\ &\Rightarrow \delta'(x) = 0 \end{aligned} \tag{B.11}$$

合并上述两种子情况，我们可以得到 $\delta'(x)$ 和 $\delta(y)$ 的关系：

$$\delta'(x) = \begin{cases} -1 & : \quad \delta(y) = 1 \\ 0 & : \quad otherwise \end{cases} \tag{B.12}$$

对于 $\delta'(z)$，根据定义，它等于：

$$\begin{aligned} \delta'(z) &= |D| - |C| && \{\delta(z) = -1 = |D| - |y|\} \\ &= |y| - |C| - 1 && \{|y| = 1 + max(|B|, |C|)\} \\ &= max(|B|, |C|) - |C| \end{aligned} \tag{B.13}$$

如果 $\delta(y) = -1$，则有 $|C| - |B| = -1$，所以 $max(|B|, |C|) = |B| = |C| + 1$。将其代入式 (B.13) 中，我们有：$\delta'(z) = 1$。

反之，如果 $\delta(y) \neq -1$，则 $max(|B|, |C|) = |C|$，我们有 $\delta'(z) = 0$。

合并上述两种子情况，$\delta'(z)$ 和 $\delta(y)$ 的关系如下：

$$\delta'(z) = \begin{cases} 1 & : \quad \delta(y) = -1 \\ 0 & : \quad otherwise \end{cases} \tag{B.14}$$

最后，对于 $\delta'(y)$，我们可以推导出下面的关系：

$$\begin{aligned} \delta'(y) &= |z| - |x| \\ &= max(|C|, |D|) - max(|A|, |B|) \end{aligned} \tag{B.15}$$

这里又分为 3 种子情况：

- 若 $\delta(y) = 0$，说明 $|B| = |C|$，根据式 (B.12) 和式 (B.14)，我们有：$\delta'(x) = 0 \Rightarrow |A| = |B|$ 以及 $\delta'(z) = 0 \Rightarrow |C| = |D|$。因此 $\delta'(y) = 0$。

- 若 $\delta(y) = 1$，根据式 (B.14)，我们有 $\delta'(z) = 0 \Rightarrow |C| = |D|$。

$$\begin{aligned}
\delta'(y) &= max(|C|, |D|) - max(|A|, |B|) && \{|C| = |D|\} \\
&= |C| - max(|A|, |B|) && \{\text{根据式 (B.12): } \delta'(x) = -1 \Rightarrow |B| - |A| = -1\} \\
&= |C| - (|B| + 1) && \{\delta(y) = 1 \Rightarrow |C| - |B| = 1\} \\
&= 0
\end{aligned}$$

- 若 $\delta(y) = -1$，根据式 (B.12)，我们有 $\delta'(x) = 0 \Rightarrow |A| = |B|$。

$$\begin{aligned}
\delta'(y) &= max(|C|, |D|) - max(|A|, |B|) && \{|A| = |B|\} \\
&= max(|C|, |D|) - |B| && \{\text{根据式 (B.14): } |D| - |C| = 1\} \\
&= |C| + 1 - |B| && \{\delta(y) = -1 \Rightarrow |C| - |B| = -1\} \\
&= 0
\end{aligned}$$

全部三种情况的结果都是 $\delta'(y) = 0$。

将上述结果归纳起来，可以得到新的平衡因子如下：

$$\begin{aligned}
\delta'(x) &= \begin{cases} -1 & : \quad \delta(y) = 1 \\ 0 & : \quad otherwise \end{cases} \\
\delta'(y) &= 0 \\
\delta'(z) &= \begin{cases} 1 & : \quad \delta(y) = -1 \\ 0 & : \quad otherwise \end{cases}
\end{aligned} \tag{B.16}$$

左－右偏（Left-right lean）的情况

左－右偏的情况和右－左偏的情况对称。使用类似的推导，我们可以得到和式 (B.16) 完全相同的结果。

B.3 删除算法

删除后会引起子树高度的变化。如果平衡因子因此超出了 $[-1, 1]$ 的范围，就需要修复以维持 AVL 树的性质。

B.3.1 函数式删除算法

我们可以复用大部分的二叉搜索树删除算法，然后检查平衡因子并进行修复。和插入算法类似，删除的结果为一对值 $(T', \Delta H)$，其中 T' 是删除后的新树、ΔH 是高

度的减少量。令函数 $first(pair)$ 返回一对值中的前一个分量。我们可以将删除算法定义如下：

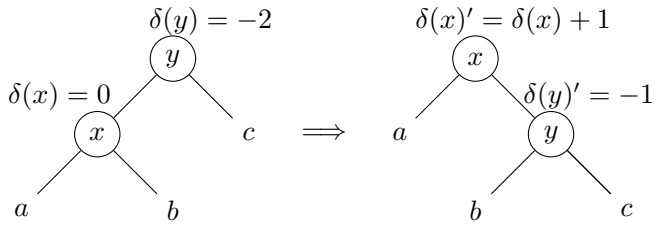
$$delete(T, k) = first(del(T, k)) \tag{B.17}$$

其中

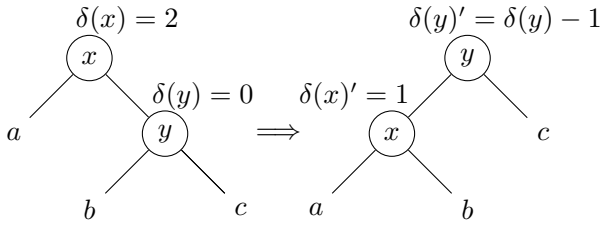
$$del(T, k) = \begin{cases} (\phi, 0) & : & T = \phi \\ tree(del(T_l, k), k', (T_r, 0), \Delta) & : & k < k' \\ tree((T_l, 0), k', del(T_r, k), \Delta) & : & k > k' \\ (T_r, -1) & : & k = k', T_l = \phi \\ (T_l, -1) & : & k = k', T_r = \phi \\ tree((T_l, 0), k'', del(T_r, k''), \Delta) & : & otherwise, k'' = min(T_r) \end{cases} \tag{B.18}$$

如果树为空，删除的结果也为空；否则，我们比较节点的 key 和待删除的值，并沿着子树递归地进行查找和删除。如果待删除的节点含有的非空子树少于两棵，可以将其直接切下。否则，我们用右子树中的最小值替换节点的 key，然后将最小值节点切下。

我们可以复用定义好的 $tree()$ 函数和 ΔH 的结果。和插入算法相比，需要处理两种删除中特有的额外情况。



(a) AVL 树删除后需要修复的情况 A



(b) AVL 树删除后需要修复的情况 B

图 B.2: AVL 树在删除后需要修复的两种情况

如图B.2所示，两种情况都可以通过一次树的旋转操作来修复。我们可以用模式

匹配来概括这种结构上的变化。

$$balance(T, \Delta H) = \begin{cases} & \cdots & : \\ (A, x, (B, y, C, -1), \delta(x) + 1, \Delta H) & : & T = ((A, x, B\delta(x)), y, C, -2, \Delta H) \\ ((A, x, B1), y, C, \delta(y) - 1, \Delta H) & : & T = (A, x, (B, y, C\delta(y)), 2, \Delta H) \\ & \cdots & : \end{cases} \tag{B.19}$$

下面的 Haskell 例子程序实现了这种 AVL 树的删除算法。

```
delete::(Ord a) ⇒ AVLTree a → a → AVLTree a
delete t x = fst $ del t x where
  -- 结果为一对值 (t, d)，t：树、d：高度的减小值
  del Empty _ = (Empty, 0)
  del (Br l k r d) x
    | x < k = node (del l x) k (r, 0) d
    | x > k = node (l, 0) k (del r x) d
    -- x == k，删除该节点
    | isEmpty l = (r, -1)
    | isEmpty r = (l, -1)
    | otherwise = node (l, 0) k' (del r k') d where k' = min r
```

其中辅助函数 `min` 和二叉搜索树中的定义相同，它沿着左子树遍历到尽头。函数 `isEmpty` 检查给定的树是否为空。

```
isEmpty Empty = True
isEmpty _ = False

min :: AVLTree a → a
min (Br Empty x _ _) = x
min (Br l _ _ _) = min l
```

在函数 `balance` 中增加用于删除的两种情况后，总共有 7 种情况需要修复。

```
balance :: (AVLTree a, Int) → (AVLTree a, Int)
balance (Br (Br (Br a x b dx) y c (-1)) z d (-2), dH) =
        (Br (Br a x b dx) y (Br c z d 0) 0, dH-1)
balance (Br a x (Br b y (Br c z d dz)    1)    2, dH) =
        (Br (Br a x b 0) y (Br c z d dz) 0, dH-1)
balance (Br (Br a x (Br b y c dy)    1) z d (-2), dH) =
        (Br (Br a x b dx') y (Br c z d dz') 0, dH-1) where
    dx' = if dy ==  1 then -1 else 0
    dz' = if dy == -1 then  1 else 0
balance (Br a x (Br (Br b y c dy) z d (-1))    2, dH) =
        (Br (Br a x b dx') y (Br c z d dz') 0, dH-1) where
    dx' = if dy ==  1 then -1 else 0
    dz' = if dy == -1 then  1 else 0
-- 删除时需要额外处理的两种情况
balance (Br (Br a x b dx) y c (-2), dH) = (Br a x (Br b y c (-1)) (dx+1), dH)
balance (Br a x (Br b y c dy)    2, dH) = (Br (Br a x b    1) y c (dy-1), dH)
balance (t, d) = (t, d)
```

B.3.2 命令式删除算法

命令式删除算法使用树的旋转操作来恢复平衡。和插入算法相比，删除需要处理更多的情况。我们首先使用二叉搜索树删除算法，然后再修复由于子树高度变化引起的平衡问题。删除算法的基本部分描述如下：

```
1: function DELETE(T, x)
2:     if x = NIL then
3:         return T
4:     p ← PARENT(x)
5:     if LEFT(x) = NIL then
6:         y ← RIGHT(x)
7:         replace x with y
8:     else if RIGHT(x) = NIL then
9:         y ← LEFT(x)
10:        replace x with y
11:    else
12:        z ← MIN(RIGHT(x))
13:        copy key and satellite date from z to x
14:        p ← PARENT(z)
15:        y ← RIGHT(z)
16:        replace z with y
17:    return AVL-DELETE-FIX(T, p, y)
```

首先时特殊情况。如果待删除节点为空，树没有变化。一般情况下，我们记待删除节点的父节点为 p，如果任一子树为空，我们将待删除节点切下，取代为另一非空子树。否则，我们在右子树中找到最小值节点 z，将其中的 key 和数据复制到待删除节点 x，然后将 z 切下。最后，我们调用修复函数，并传入根节点、父节点、和替换掉待删除节点的新节点。

记父节点的平衡因子为 $\delta(p)$，和删除后的新平衡因子 $\delta(p)'$ 比较，我们会发现三种不同的情况。

- 情况 1：$|\delta(p)| = 0$、$|\delta(p)'| = 1$。这说明，虽然删除后子树的高度降低了，但是父节点仍然满足 AVL 树的性质，保持平衡。算法退出；

- 情况 2：$|\delta(p)| = 1$、$|\delta(p)'| = 0$。这说明，删除前左右子树的高度差为 1，删除后原来较高的树高度减小了 1。左右子树现在高度相等。结果是父节点的高度也减小了 1。我们需要继续自底向上沿着父节点更新树的高度；

- 情况 3：$|\delta(p)| = 1$、$|\delta(p)'| = 2$。这说明删除后子树的高度差违反了 AVL 树的性质，我们需要通过树的旋转操作来修复平衡。

对于情况 3，大部分的修复操作和插入算法相同。但是我们需要针对图B.2中所示的两种特殊情况，进行额外的处理。完整的修复算法如下。

```
1: function AVL-Delete-Fix(T, p, x)
2:     while p ≠ NIL do
3:         l ← Left(p), r ← Right(p)
4:         δ ← δ(p), δ' ← δ
5:         if x = l then
6:             δ' ← δ' + 1
7:         else
8:             δ' ← δ' − 1
9:         if p is leaf then                                  ▷ l = r = NIL
10:            δ' ← 0
11:        if |δ| = 1 ∧ |δ'| = 0 then
12:            x ← p
13:            p ← Parent(x)
14:        else if |δ| = 0 ∧ |δ'| = 1 then
15:            return T
16:        else if |δ| = 1 ∧ |δ'| = 2 then
17:            if δ' = 2 then
18:                if δ(r) = 1 then                           ▷ 右－右情况
19:                    δ(p) ← 0
20:                    δ(r) ← 0
21:                    p ← r
22:                    T ← Left-Rotate(T, p)
23:                else if δ(r) = −1 then                     ▷ 右－左情况
24:                    δ_y ← δ( Left(r) )
25:                    if δ_y = 1 then
26:                        δ(p) ← −1
27:                    else
28:                        δ(p) ← 0
29:                    δ( Left(r) ) ← 0
30:                    if δ_y = −1 then
31:                        δ(r) ← 1
32:                    else
33:                        δ(r) ← 0
34:                else                                       ▷ 删除特有的右－右情况
35:                    δ(p) ← 1
```

```
36:                 δ(r) ← δ(r) − 1
37:                 T ← LEFT-ROTATE(T, p)
38:                 break                                   ▷ 高度不再变化
39:             else if δ′ = −2 then
40:                 if δ(l) = −1 then                       ▷ 左－左情况
41:                     δ(p) ← 0
42:                     δ(l) ← 0
43:                     p ← l
44:                     T ← RIGHT-ROTATE(T, p)
45:                 else if δ(l) = 1 then                   ▷ 左－右情况
46:                     δ_y ← δ( RIGHT(l) )
47:                     if δ_y = −1 then
48:                         δ(p) ← 1
49:                     else
50:                         δ(p) ← 0
51:                     δ( RIGHT(l) ) ← 0
52:                     if δ_y = 1 then
53:                         δ(l) ← −1
54:                     else
55:                         δ(l) ← 0
56:                 else                                    ▷ 删除特有的左－左情况
57:                     δ(p) ← −1
58:                     δ(l) ← δ(l) + 1
59:                     T ← RIGHT-ROTATE(T, p)
60:                     break                               ▷ 高度不再继续变化
                                                            ▷ 高度减小，继续自底向上更新
61:             x ← p
62:             p ← PARENT(x)
63:     if p = NIL then                                     ▷ 删除根节点
64:         return x
65:     return T
```

下面的 C++ 例子程序实现了 AVL 树的删除算法。

```
Node* del(Node* t, Node* x) {
    if (!x) return t;
    Node *y, *parent = x→parent;
    if (!x→left) {
        y = x→replaceWith(x→right);
    } else if (!x→right) {
        y = x→replaceWith(x→left);
    } else {
```

```
        y = min(x→right);
        x→key = y→key;
        parent = y→parent;
        x = y;
        y = y→replaceWith(y→right);
    }
    t = deleteFix(t, parent, y);
    remove(x);
    return t;
}
```

其中方法 replaceWith(tree) 用传入的子树替换掉当前节点，它返回新节点作为结果。

```
Node* replaceWith(Node* y) {
    return replace(parent, this, y);
}
```

函数 replace(parent, x, y) 用子树 y 替换掉父节点的子树 x。

```
// change from: parent -→ x to parent -→ y
Node* replace(Node* parent, Node* x, Node* y) {
    if (!parent) {
        if (y) y→parent = nullptr;
    } else if (parent→left == x) {
        parent→setLeft(y);
    } else {
        parent→setRight(y);
    }
    if (x) x→parent = nullptr;
    return y;
}
```

函数 min(t) 递归地查找子树的最小值节点。函数 remove(tree) 用于将节点释放。

```
Node* min(Node* t) {
    while (t && t→left) t = t→left;
    return t;
}

void remove(Node* x) {
    if (x) {
        x→parent = x→left = x→right = nullptr;
        delete x;
    }
}
```

修复函数的实现如下。

```
Node* deleteFix(Node* t, Node* parent, Node* x) {
    int d1, d2, dy;
    Node *p, *l, *r;
```

```
    while (parent) {
        d2 = d1 = parent→delta;
        d2 += (x == parent→left ? 1 : -1);
        if (isLeaf(parent)) d2 = 0;
        parent→delta = d2;
        p = parent;
        l = parent→left;
        r = parent→right;
        if (abs(d1) == 1 && abs(d2) == 0) {
            x = parent;
            parent = x→parent;
        } else if (abs(d1) == 0 && abs(d2) == 1) {
            return t;
        } else if (abs(d1) == 1 && abs(d2) == 2) {
            if (d2 == 2) {
                if (r→delta == 1) {  // 右－右情况
                    p→delta = r→delta = 0;
                    parent = r;
                    t = leftRotate(t, p);
                } else if (r→delta == -1) { // 右－左情况
                    dy = r→left→delta;
                    p→delta = dy == 1 ? -1 : 0;
                    r→left→delta = 0;
                    r→delta = dy == -1 ? 1 : 0;
                    parent = r→left;
                    t = rightRotate(t, r);
                    t = leftRotate(t, p);
                } else { // 删除特有的右－右情况
                    p→delta = 1;
                    r→delta--;
                    t = leftRotate(t, p);
                    break; // 高度不再继续变化
                }
            } else if (d2 == -2) {
                if (l→delta == -1) { // 左－左情况
                    p→delta = l→delta = 0;
                    parent = l;
                    t = rightRotate(t, p);
                } else if (l→delta == 1) { // 左－右情况
                    dy = l→right→delta;
                    l→delta = dy == 1 ? -1 : 0;
                    l→right→delta = 0;
                    p→delta = dy == -1 ? 1 : 0;
                    parent = l→right;
                    t = leftRotate(t, l);
                    t = rightRotate(t, p);
                } else { // 删除特有的左－左情况
                    p→delta = -1;
                    l→delta++;
                    t = rightRotate(t, p);
                    break; // 高度不再继续变化
                }
            }
```

```
            // 以上修复会造成高度减小，需要继续自底向上更新
            x = parent;
            parent = x→parent;
        } else {
            printf("shouldn't be here, d1 = %d, d2 = %d", d1, d2);
            assert(false);
        }
    }
    if (!parent) return x; // 删除根节点
    return t;
}
```

练习 B.1

比较 AVL 树的命令式删除和插入算法，将共同的部分抽出，实现一个通用的 AVL 树修复算法。

附录 C　后缀树

C.1　简介

后缀树是一种重要的数据结构，它可以用来实现很多快速字符串操作算法 [23]。后缀树还在生物信息处理中被广泛用于 DNA 模式匹配 [29]。Weiner 在 1973 年最早引入了后缀树 [28]，最新的 on-line 构造算法发现于 1995 年 [27]。

字符串 S 的后缀树是一棵特殊的 Patricia（见上一章 Radix 树的介绍）。树中的所有边都由 S 的某个子串标记。S 的每个后缀都唯一对应一条从根到叶子的路径。图C.1显示了一棵对应英文单词‘banana’的后缀树。

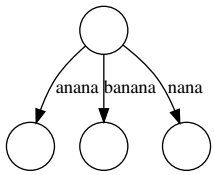


图 C.1: 英文“banana”对应的后缀树

所有的后缀“banana”、“anana”、“nana”、“ana”、“na”、“a”和空串“”都可以在上面的后缀树中找到。这些后缀中，前三个存在明确的对应路径，其余的并未明确显示出来。这是因为后面四个后缀：“ana”、“na”、“a”和空串“”同时也是其他后缀的前缀。为了将所有后缀都明确显示出来，我们可以在原字符串的末尾附加一个特殊的终结符，这一终结符不在字符串的其他位置出现。我们通常把它标记为‘$’。这样，就不存在任何后缀同时也是其他某个后缀的前缀。

虽然字符串“banana”的后缀树很简单，但是字符串“bananas”的后缀树却大相径庭。如图C.2所示。

我们可以复用上一章中介绍过的 Patricia 插入算法来构造后缀树。

```
1: function SUFFIX-TREE(S)
2:     T ← NIL
3:     for i ← 1 to |S| do
4:         T ← PATRICIA-INSERT(T, RIGHT(S, i))
5:     return T
```

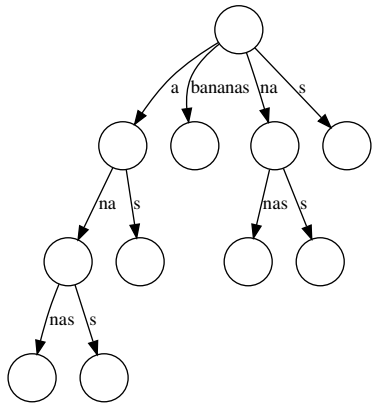


图 C.2: 字符串“bananas”对应的后缀树

对于非空字符串 $S = s_1 s_2 ... s_i ... s_n$，长度 $n = |S|$，函数 $\text{RIGHT}(S, i) = s_i s_{i+1} ... s_n$。它的结果是一个子串，从 S 的第 i 个字符直到末尾。这一直观的算法也可以定义如下：

$$suffix_T(S) = fold(insert_{Patricia}, \phi, suffixes(S)) \tag{C.1}$$

其中函数 $suffixes(S)$ 枚举字符串 S 的所有后缀。如果字符串为空，结果是一个空串；否则 S 本身是自己的一个后缀，其余后缀可以通过递归调用 $suffixes(S')$ 来获取。这里 S' 是 S 除去第一个字符以外的其余部分。

$$suffixes(S) = \begin{cases} \{\phi\} & : \quad S = \phi \\ \{S\} \cup suffixes(S') & : \quad otherwise \end{cases} \tag{C.2}$$

对于长度为 n 的字符串，这一方法需要 $O(n^2)$ 的时间来构造后缀树。这是因为它总共将 n 个后缀插入到树中，而每次插入的时间和后缀的长度成正比。算法的性能不够好。

本章中，我们首先介绍一种快速后缀 Trie 的构造方法，它使用了后缀链接（suffix link）的概念。由于 Trie 耗费大量的空间，我们接下来介绍一种由 Ukkonen 发现的在线性时间内 on-line 构造后缀树的算法。最后，我们介绍如何使用后缀树来解决一些有趣的字符串处理问题。

C.2 后缀 Trie

如同 Trie 和 Patricia 的关系，后缀 Trie 比后缀树的结构简单许多。图C.3是英文单词“banana”对应的后缀 Trie。

和图C.1比较，我们可以发现后缀树和后缀 Trie 的区别。后缀 Trie 中的每条边仅仅代表一个字符，而不是一个子串。因此后缀 Trie 需要使用更多的空间来存储信息。如果我们将只含有一个子树的节点压缩到一起，后缀 Trie 就转变成一棵后缀树。

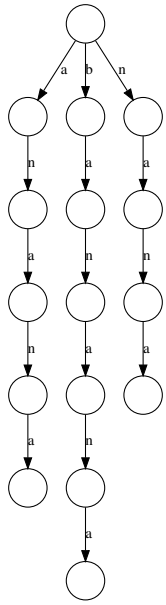


图 C.3: “banana”对应的后缀 Trie

我们可以复用 Trie 的定义：每个节点对应一个字符，节点包含多个子树。子树可以通过对应的字符来引用。

C.2.1 节点转移和后缀链接

设字符串 S 的长度为 n，定义 $S_i = s_1 s_2 ... s_i$ 为包含前 i 个字符的前缀。

在后缀 Trie 中，每个节点都代表一个后缀。如图C.4所示的例子，节点 X 代表后缀“a”，通过添加字符‘c’，节点 X 转移到节点 Y，节点 Y 代表后缀“ac”。我们称节点 X 通过字符‘c’所代表的边转移到节点 Y[27]。

$Y \leftarrow \textsc{Children}(X)[c]$

同时，我们也称节点 X 有一个‘c’子节点 Y。下面的 Python 表达式给出了一个节点转移的一个例子。

```
y = x.children[c]
```

如果后缀 Trie 中的节点 A 代表后缀 $s_i s_{i+1} ... s_n$，节点 B 代表后缀 $s_{i+1} s_{i+2} ... s_n$，我们称节点 B 代表节点 A 的后缀。我们可以创建一个从 A 到 B 的链接，称为节点 A 的后缀链接[27]。我们通常用虚线箭头来表示后缀链接。在图C.4的例子中，节点 A 的后缀链接指向节点 B，节点 B 的后缀链接指向节点 C。

除根节点外，所有的节点都存在后缀链接。为此我们在 Trie 的定义中增加一个后缀链接的部分，如下面的 Python 例子代码所示。

```
class STrie:
    def __init__(self, suffix=None):
        self.children = {}
```

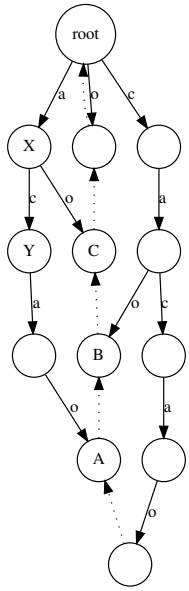


图 C.4: 字符串‘cacao’对应的后缀 Trie。节点 $X \leftarrow$ “a”、节点 $Y \leftarrow$ “ac”，节点 X 通过字符‘c’转移到 Y。

```
        self.suffix = suffix
```

C.2.2 On-line 构造

对于字符串 S,假设我们已经构造了第 i 个前缀 $S_i = s_1 s_2 ... s_i$ 的后缀 Trie。记这一后缀 Trie 为 $SuffixTrie(S_i)$。我们考虑如何从 $SuffixTrie(S_i)$ 获得 $SuffixTrie(S_{i+1})$。

如果列出 $SuffixTrie(S_i)$ 中的全部后缀，按照从最长的（就是 S_i 本身）到最短的（为空串）的顺序，我们可以获得表C.1。总共有 $i+1$ 个后缀。

后缀
$s_1 s_2 s_3 ... s_i$
$s_2 s_3 ... s_i$
...
$s_{i-1} s_i$
s_i
“”

表 C.1: S_i 的全部后缀

我们可以向表中的每个后缀后面添加字符 s_{i+1}，然后再增加一个空串。这样就获得了 S_{i+1} 的全部后缀。这等效于给 Trie 中的所有节点增加一个代表字符 s_{i+1} 的新节点。

Algorithm 5 从 $SuffixTrie(S_i)$ 获取 $SuffixTrie(S_{i+1})$，最初的版本

```
1: for ∀T ∈ SuffixTrie(S_i) do
2:     CHILDREN(T)[s_{i+1}] ← CREATE-EMPTY-NODE
```

但是，$SuffixTrie(S_i)$ 中的某些节点可能已经有 s_{i+1} 子节点了。例如，图C.5中，节点 X 和节点 Y 分别代表后缀“cac”和“ac”。它们没有‘a’子节点；但是代表后缀“c”的节点 Z，已经有‘a’子节点了。

当向 $SuffixTrie(S_i)$ 增加字符 s_{i+1}（这里 s_{i+1} 为‘a’）时，我们需要为 X 和 Y 新建子节点，但是我们不需要给 Z 新建子节点。

如果我们逐一检查表C.1中的每一项，当发现一个节点已经有 s_{i+1} 子节点时，我们可以立即停止。这是因为，如果 $SuffixTrie(S_i)$ 中的节点 X 已经有 s_{i+1} 子节点，根据后缀链接的定义，$SuffixTrie(S_i)$ 中任何 X 的后缀节点 X' 一定也存在 s_{i+1} 子节点。也就是说，设 $c = s_{i+1}$，若 wc 是 S_i 的子串，则 wc 的每个前缀也都是 S_i 的子串 [27]。唯一的例外是根节点，因为根节点代表空串“”。

根据上面的分析，我们可以将算法5改进成6。

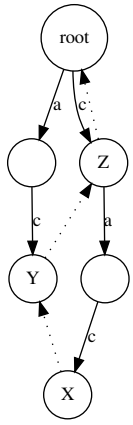
Algorithm 6 从 $SuffixTrie(S_i)$ 获取 $SuffixTrie(S_{i+1})$，改进版本

```
1: for each T ∈ SuffixTrie(S_i) 按照后缀长度递减顺序 do
2:     if CHILDREN(T)[s_{i+1}] = NIL then
3:         CHILDREN(T)[s_{i+1}] ← CREATE-EMPTY-NODE
4:     else
5:         break
```

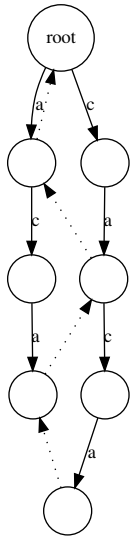
接下来的问题是如何按照后缀的长度，从长到短降序遍历所有的节点？我们定义一棵后缀 Trie 的top 为深度最大的叶子节点。这一定义保证了 top 指向最长的后缀。我们从 top 开始，沿着后缀链接每前进一步，后缀的长度就相应减一。沿着后缀链接我们可以一直从 top 遍历到 root。这一遍历的顺序恰好符合我们的要求。最后，我们需要处理一棵特殊的 Trie $SuffixTrie(NIL)$，它对应空串。这种情况下，我们定义 top 和 root 相等。

函数 INSERT 从 $SuffixTrie(S_i)$ 构造 $SuffixTrie(S_{i+1})$。它接受两个参数：一个是 $SuffixTrie(S_i)$ 的 top 位置，另外一个是字符 s_{i+1}。如果 top 为空（NIL），说明树也是空的，根节点 root 也就不存在。这种情况下我们需要创建一个 root。我们使用一个空节点 T' 作为哨兵（sentinel）节点。它可以用来记录上一次创建的新节点。在主循环中，算法沿着后缀链接逐一检查每个节点。如果 s_{i+1} 子节点不存在，就创建一个新节点，并将其对应到字符 s_{i+1} 上。算法不断沿着后缀链接遍历直到根节点 root，或者中途遇到一个已经有 s_{i+1} 子节点的位置。这时，最后一个后缀链接指向这一子节点。最后算法返回新的 top 位置用于将后继字符插入到后缀 Trie 中。

给定字符串 S，我们可以通过不断调用 INSERT 函数来构造后缀 Trie。



(a) 字符串“cac”对应的后缀 Trie。



(b) 字符串“caca”对应的后缀 Trie。

图 C.5: 字符串“cac”和“caca”对应的后缀 Trie

```
function INSERT(top, c)
    if top = NIL then                                    ▷ Trie 为空
        top ← CREATE-EMPTY-NODE
    T ← top
    T' ← CREATE-EMPTY-NODE                               ▷ 用 dummy 值初始化
    while T ≠ NIL ∧ CHILDREN(T)[c] = NIL do
        CHILDREN(T)[c] ← CREATE-EMPTY-NODE
        SUFFIX-LINK(T') ← CHILDREN(T)[c]
        T' ← CHILDREN(T)[c]
        T ← SUFFIX-LINK(T)
    if T ≠ NIL then
        SUFFIX-LINK(T') ← CHILDREN(T)[c]
    return CHILDREN(top)[c]                              ▷ 返回新的 top 节点
```

```
1: function SUFFIX-TRIE(S)
2:     t ← NIL
3:     for i ← 1 to |S| do
4:         t ← INSERT(t, s_i)
5:     return t
```

注意：这一算法的返回值不是根节点，而是后缀 Trie 的 top 节点。我们可以沿着后缀链接遍历到根节点。

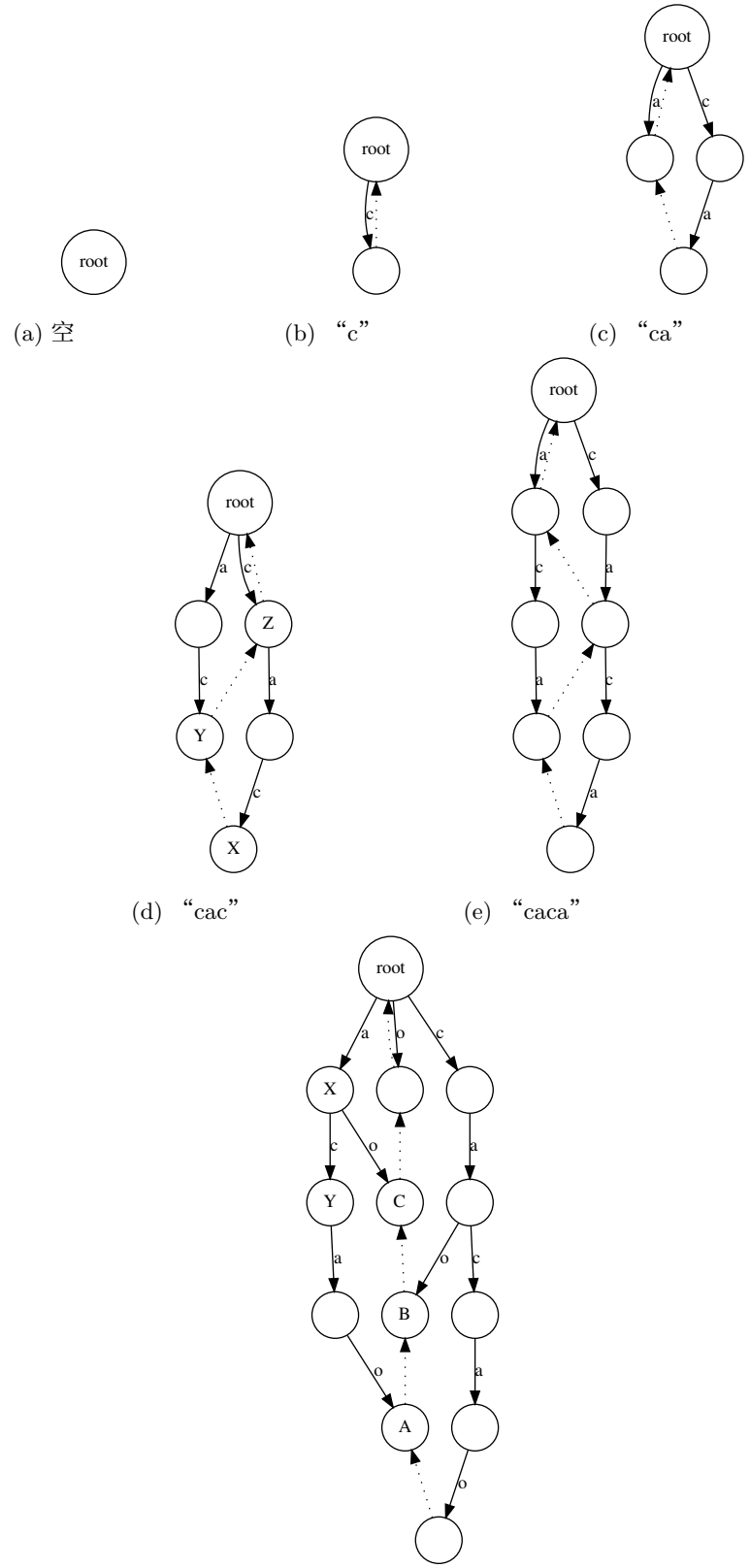
```
1: function ROOT(T)
2:     while SUFFIX-LINK(T) ≠ NIL do
3:         T ← SUFFIX-LINK(T)
4:     return T
```

图C.6给出了构造字符串“cacao”后缀树的步骤。简单起见，我们只画出了最后一组后缀链接。

由于每次沿着后缀链接遍历，算法 INSERT 的时间复杂度和后缀 Trie 的大小成正比。最坏情况下，对于长度为 n 的字符串，需要 $O(n^2)$ 时间来构造后缀 Trie。例如字符串 $S = a^n b^n$，有 n 个字符 a 和 n 个字符 b，就会使得算法的性能严重下降。

下面的 Python 例子程序实现了后缀 Trie 的构造算法。

```
def suffix_trie(str):
    t = None
    for c in str:
        t = insert(t, c)
    return root(t)

def insert(top, c):
```



(a) 空 (b) “c” (c) “ca”

(d) “cac” (e) “caca”

(f) “cacao”

图 C.6: 构造字符串“cacao”的后缀 Trie。共分 6 步。虚线箭头表示最后一组后缀链接。

```
    if top is None:
        top=STrie()
    node = top
    new_node = STrie() #用 dummy 值初始化
    while (node is not None) and (c not in node.children):
        new_node.suffix = node.children[c] = STrie(node)
        new_node = node.children[c]
        node = node.suffix
    if node is not None:
        new_node.suffix = node.children[c]
    return top.children[c] #更新 top 节点

def root(node):
    while node.suffix is not None:
        node = node.suffix
    return node
```

C.3 后缀树

后缀 Trie 既耗费空间，构造起来又慢（平方时间复杂度）。有些实现仅仅将构后缀 Trie 压缩成后缀树 [30]，但是这样还不够理想。Ukkonen 在 1995 年发表了一种高效的 on-line 构造算法，可以在线性时间内构造后缀树。

C.3.1 on-line 构造

为了实现线性时间的后缀树构造算法，Ukkonen 引入了两个重要的概念：活动点（active point）和终止点（end point），并使用引用对来减少对空间的占用，本节我们将依次介绍这些内容。

活动点（active point）和终止点（end point）

在后缀 trie 构造算法中，我们可以从任意一个中间结果 $SuffixTrie(S_i)$ 得到下一个结果 $SuffixTrie(S_{i+1})$。这一点很具有启发性。我们观察一下图C.6中的最后两步。

一共有两种不同的更新：

1. 所有的叶子节点都添加了一个代表字符 s_{i+1} 的新节点；
2. 某些中间节点被分支出一个代表字符 s_{i+1} 的新节点。

其中第一种更新简单易懂，我们总是要为下一个字符增加新节点。Ukkonen 将所有的叶子节点定义为“开放节点”。

我们更加关心第二种更新。什么样的中间节点会被分支出新节点？我们希望能快速定位到它们，以实施更新。

Ukkonen 将从 top 沿着后缀链接前进的路径定义为“boundary 路径”。记 boundary 路径上的各个节点为 $n_1, n_2, ..., n_j, ..., n_k$。显然，第一个是叶子节点（为 top），假设从第 j 个节点开始不再是叶子节点，此后我们需要不断分支出新节点直到处理完第 k 个。

Ukkonen 将此路径上的第一个非叶子节点 n_j 定义为活动点（active point），最后一个节点 n_k 定义为终止点（end point），它有可能是根节点 root。

引用对（Reference pair）

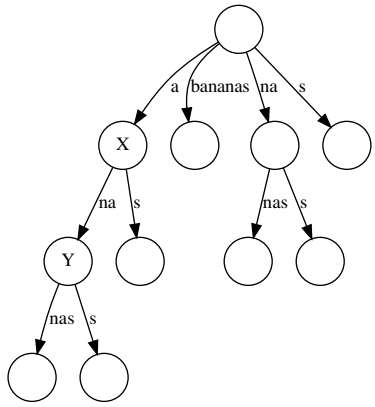


图 C.7: 字符串“bananas”的后缀树。节点 X 通过子串“na”转移到节点 Y。

图C.7是字符串“bananas”对应的后缀树。节点 X 代表后缀“a”。通过添加子串“na”，节点 X 转移到代表后缀“ana”的节点 Y。也就是说，我们可以将 Y 表达为一对值，由一个节点和一个子串组成。形如 (X, w)，其中 $w =$“na”。

Ukkonen 称这样的一对值为引用对（reference pair）。不光所有的可见节点，而且连不可见的隐藏节点都可以用引用对来表示。例如 (X，“n”) 就表达了图C.7中一个隐藏节点的位置。通过使用引用对，我们可以定位到后缀树中的任何位置。

给定字符串 S，它的所有子串都可以用一对索引 (l, r) 来确定。其中，l 是子串最左侧字符的索引，而 r 是最右侧字符的索引。例如，若字符串 $S =$“bananas”，并且索引从 1 开始，子串“na”可以表达为一对索引 $(3, 4)$。通过使用索引对，我们可以仅仅保留一份字符串的完整拷贝，从而大量节省空间。后缀树中的任何位置都可以定义为形如 $(node, (l, r))$ 的表达式。这就是引用对的最终形式。

利用引用对，后缀树中的节点转移可以定义如下：

$$\textsc{Children}(X)[s_l] \leftarrow ((l, r), Y) \iff Y \leftarrow (X, (l, r))$$

若字符 $s_l = c$，我们称节点 X 有一个 c 子节点 Y。Y 可以由 X 通过子串 (l, r) 转移到。每个节点最多有一个 c 子节点。

归一化引用对

显然,后缀树中的一个位置可能存在多个引用对。例如图C.7中的节点 Y 既可以表示为引用对 $(X,(3,4))$, 也可以表示为 $(root,(2,4))$。如果我们定义空串为 $\epsilon=(i,i-1)$, Y 还可以表示为 (Y,ϵ)。

所谓归一化引用对，是指含有最近节点的引用对。特别地，对于后缀树中的某个节点，归一化引用对由该节点和空串组成。所以节点 Y 的归一化引用对为 (Y,ϵ)。

下面的算法将任一引用对 $(node,(l,r))$ 转换为归一化引用对 $(node',(l',r))$。由于转换后 r 不会变，所以算法仅仅返回 $(node',l')$ 作为结果。

Algorithm 7 将任一引用对转换为归一化引用对

```
1:  function CANONIZE(node, (l, r))
2:      if node = NIL then
3:          if (l, r) = ϵ then
4:              return ( NIL, l)
5:          else
6:              return CANONIZE(root, (l + 1, r))
7:      while l ≤ r do                                  ▷ (l, r) 不为空
8:          ((l', r'), node') ← CHILDREN(node)[s_l]
9:          if r − l ≥ r' − l' then
10:             l ← l + r' − l' + 1                     ▷ 从 (l, r) 去除掉 |(l', r')| 个字符
11:             node ← node'
12:         else
13:             break
14:     return (node, l)
```

算法需要单独处理传入空节点 NIL 的情况。此时算法必定按照下面的方式调用：

CANONIZE(SUFFIX-LINK($root$), (l,r))

由于根节点的后缀链接为空 NIL,若子串 (l,r) 不等于 ϵ,则结果为 $(root,(l+1,r))$; 否则，我们应该返回一个特殊的结束位置：(NIL, ϵ)。

我们将稍后详细解释这一特殊情况。

Ukkonen 算法

我们在前面C.3.1一节中提到，为所有的叶子添加代表新字符的节点很简单。使用引用对的概念，当我们从后缀树 $SuffixTree(S_i)$ 更新到 $SuffixTree(S_{i+1})$ 时，所有形如 $(node,(l,i))$ 的节点都是叶子节点。它们将更新为 $(node,(l,i+1))$。Ukkonen 为此将叶子表示为 $(node,(l,\infty))$, 其中无穷 ∞ 的含义是“增长开放”(open to grow)。在后缀树构造过程中，我们可以暂时忽略全部的叶子节点。当构造完毕后，只要把引用对中的无穷 ∞ 替换为字符串的长度就可以了。

这样，算法仅仅关注从活动点到终止点这条路径上的所有位置（注意：不是节点）。最关键的问题是如何找到活动点和终止点。

当开始构造后缀树时，仅仅存在一个根节点。没有任何分支和叶子。因此活动点为 $(root, \epsilon)$，或者表示为 $(root, (1, 0))$（字符串从 1 开始索引）。

而终止点，它是更新 $SuffixTree(S_i)$ 过程停止时的位置。根据后缀 Trie 算法，我们知道在它的位置 上，已经存在 s_{i+1} 子节点了。注意：后缀 Trie 中的某个节点不一定是后缀树中的可见节点。若 $(node, (l, r))$ 是终止点，则可能存在两种情况：

1. $(l, r) = \epsilon$。这说明终止点是一个可见节点。该节点含有一个 s_{i+1} 子节点，亦即 $\text{CHILDREN}(node)[s_{i+1}] \neq \text{NIL}$；
2. 否则，$l \leq r$，说明终止点是一个隐藏的位置。我们有 $s_{i+1} = s_{l'+|(l,r)|}$，$\text{CHILDREN}(node)[s_l] = ((l', r'), node')$，这里 $|(l, r)|$ 表示子串 (l, r) 的长度。它等于 $r - l + 1$。图C.8描述了这一情况。我们也可以说：$(node, (l, r))$ 隐藏含有一个 s_{i+1} 子节点。

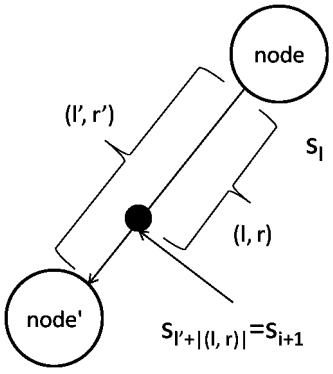


图 C.8: 隐藏终止点（end point）

Ukkonen 发现了一个重要的事实：若 $(node, (l, i))$ 是后缀树 $SuffixTree(S_i)$ 的终止点，则 $(node, (l, i + 1))$ 一定是后缀树 $SuffixTree(S_{i+1})$ 的活动点。

这是因为，如果 $(node, (l, i))$ 是后缀树 $SuffixTree(S_i)$ 的终止点，它必然含有一个 s_{i+1} 子节点（可见的或者隐藏的）。如果终止点代表后缀 $s_k s_{k+1} ... s_i$，它一定是后缀树 $SuffixTree(S_i)$ 中最长的一个同时满足 $s_k s_{k+1} ... s_i s_{i+1}$ 是 S_i 的某个子串的后缀。考虑 S_{i+1}，由于后缀 $s_k s_{k+1} ... s_i s_{i+1}$ 同时也是某个子串，它必然在 S_{i+1} 中至少出现两次，因此位置 $(node, (l, i + 1))$ 为后缀树 $SuffixTree(S_{i+1})$ 的活动点。图C.9给出了对应的解释。

总结以上各点，Ukkonen 的 on-line 构造算法可以定义如下：

```
1: function UPDATE(node, (l, i))
2:     prev ← CREATE-EMPTY-NODE                ▷ 初始化为哨兵（sentinel）节点
3:     loop                                    ▷ 沿 suffix links 遍历
```

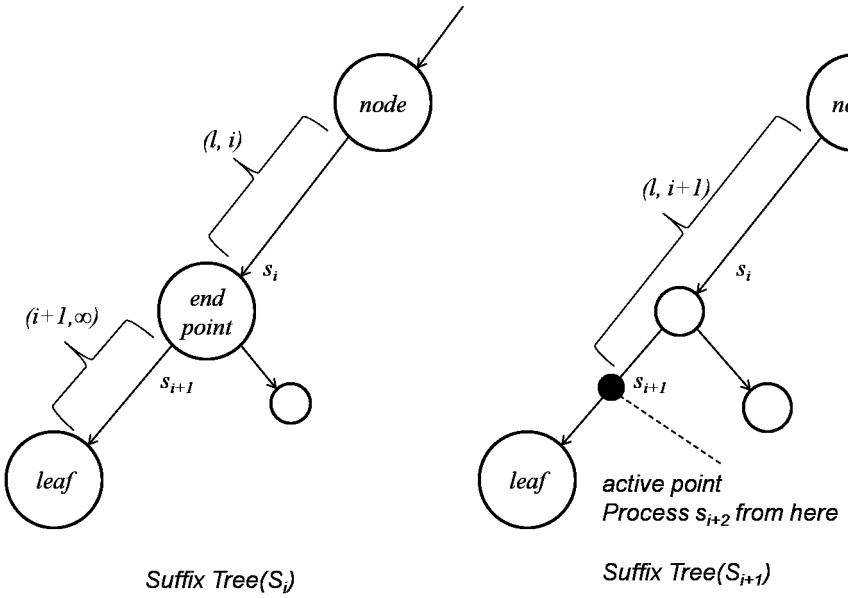


图 C.9: 后缀树 $SuffixTree(S_i)$ 的终止点（end point）和后缀树 $SuffixTree(S_{i+1})$ 的活动点（active point）

```
4:        (finish, node') ← End-Point-Branch?(node, (l, i − 1), s_i)
5:        if finish then
6:            break
7:        Children(node')[s_i] ← ((i, ∞), Create-Empty-Node)
8:        Suffix-Link(prev) ← node'
9:        prev ← node'
10:       (node, l) ← Canonize(Suffix-Link(node), (l, i − 1))
11:   Suffix-Link(prev) ← node
12:   return (node, l)                                  ▷ 返回终止点
```

算法接受一个引用对 $(node, (l, i))$ 作为参数。这里 $(node, (l, i-1)$ 是后缀树 $SuffixTree(S_{i-1})$ 的活动点。算法不断沿着后缀链接处理节点，直到位置 $(node, (l, i-1))$ 成为终止点。否则，函数 End-Point-Branch? 返回一个用于分支出新的叶子节点的位置。它的实现如下：

```
function End-Point-Branch?(node, (l, r), c)
    if (l, r) = ϵ then
        if node = NIL then
            return (TRUE, root)
        else
            return (Children(node)[c] = NIL, node)
    else
```

$((l', r'), node') \leftarrow \text{CHILDREN}(node)[s_l]$
$pos \leftarrow l' + |(l, r)|$
if $s_{pos} = c$ **then**
  return (TRUE, $node$)
else
  $p \leftarrow$ CREATE-EMPTY-NODE
  $\text{CHILDREN}(node)[s_{l'}] \leftarrow ((l', pos - 1), p)$
  $\text{CHILDREN}(p)[s_{pos}] \leftarrow ((pos, r'), node')$
  return (FALSE, p)

如果传入的位置是 $(root, \epsilon)$，说明我们到达了根节点。根节点一定是终止点，本轮的处理即可结束。如果传入的位置形如 $(node, \epsilon)$，此引用对代表一个可见节点，我们可以检查它是否已经含有一个 $c = s_i$ 的子节点。如果不含有，我们需要分支出一个新的叶子节点。

如果传入的不是一个可见节点的位置，也就是说 $(node, (l, r))$ 指向一个隐藏节点。我们需要找到它的下一个位置以判断是否含有一个 c 子节点。如果含有，说明这是一个终止点，我们可以结束本轮更新；否则，我们将此位置转化为一个可见节点，用于接下来分支出新的子节点。

Ukkonen 算法最后实现如下：

1: **function** SUFFIX-TREE(S)
2: $root \leftarrow$ CREATE-EMPTY-NODE
3: $node \leftarrow root, l \leftarrow 0$
4: **for** $i \leftarrow 1$ to $|S|$ **do**
5: $(node, l) \leftarrow \text{UPDATE}(node, (l, i))$
6: $(node, l) \leftarrow \text{CANONIZE}(node, (l, i))$
7: **return** $root$

图C.10给出了构造字符串“cacao”的后缀树的各个步骤。

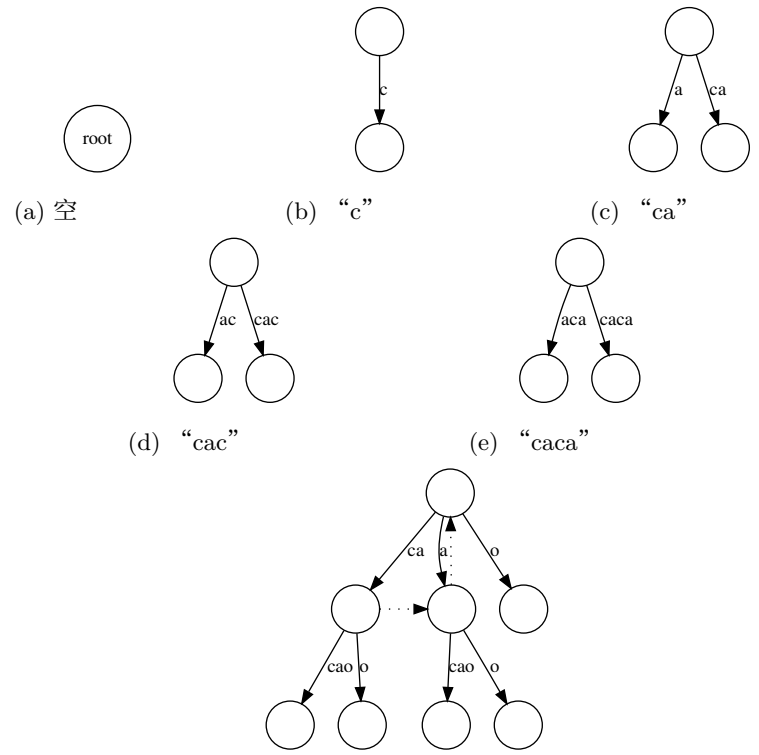
我们并不需要为叶子节点设置后缀链接，只有分支节点需要后缀链接。

下面的 Python 例子程序实现了 Ukkonen 算法。首先是节点的定义：

```
class Node:
    def __init__(self, suffix=None):
        self.children = {} # 'c':(word, Node), 其中: word = (l, r)
        self.suffix = suffix
```

为了节省空间，程序仅仅保留了一份完整的字符串，所有的子串都由左右边界对 $(left, right)$ 来表示。对于叶子节点，右侧边界是开放的，用 $(left, \infty)$ 来表示。后缀树的定义为：

```
class STree:
    def __init__(self, s):
        self.str = s
```



(a) 空 (b) “c” (c) “ca”

(d) “cac” (e) “caca”

(f) “cacao”

图 C.10: 构造字符串“cacao”的后缀树。共有 6 步。虚线箭头显示了最后一层的后缀链接。

```
        self.infinity = len(s)+1000
        self.root = Node()
```

在实际的程序中，使用完整字符串的长度加一个大数来表示无穷。下面是一些辅助函数：

```
def substr(str, str_ref):
    (l, r)=str_ref
    return str[l:r+1]

def length(str_ref):
    (l, r)=str_ref
    return r-l+1
```

Ukkonen 算法的主函数实现如下：

```
def suffix_tree(str):
    t = STree(str)
    node = t.root # 活动点初始为 (root, Empty)
    l = 0
    for i in range(len(str)):
        (node, l) = update(t, node, (l, i))
        (node, l) = canonize(t, node, (l, i))
    return t

def update(t, node, str_ref):
    (l, i) = str_ref
    c = t.str[i] # 当前字符
    prev = Node() # 用 dummy 值初始化
    while True:
        (finish, p) = branch(t, node, (l, i-1), c)
        if finish:
            break
        p.children[c]=((i, t.infinity), Node())
        prev.suffix = p
        prev = p
        (node, l) = canonize(t, node.suffix, (l, i-1))
    prev.suffix = node
    return (node, l)

def branch(t, node, str_ref, c):
    (l, r) = str_ref
    if length(str_ref)<=0: # (node, empty)
        if node is None: # '_|_'
            return (True, t.root)
        else:
            return ((c in node.children), node)
    else:
        ((l1, r1), node1) = node.children[t.str[l]]
        pos = l1+length(str_ref)
        if t.str[pos]==c:
            return (True, node)
        else:
```

```
            branch_node = Node()
            node.children[t.str[l1]]=((l1, pos-1), branch_node)
            branch_node.children[t.str[pos]] = ((pos, r1), node1)
            return (False, branch_node)

def canonize(t, node, str_ref):
    (l, r) = str_ref
    if node is None:
        if length(str_ref)≤0:
            return (None, l)
        else:
            return canonize(t, t.root, (l+1, r))
    while l≤r: # str_ref 不为空
        ((l1, r1), child) = node.children[t.str[l]]
        if r-l ≥ r1-l1:
            l += r1-l1+1
            node = child
        else:
            break
    return (node, l)
```

函数式构造后缀树

Giegerich 和 Kurtz 发现 Ukkonen 算法可以转化为 McCreight 算法 [31]。Weiner、McCreight 和 Ukkonen 分别发现的三种后缀树构造算法都是线性时间复杂度 $O(n)$ 的。Giegerich 和 Kurtz 进一步猜想，任何后缀树的顺序构造算法如果不使用后缀链接（suffix link）或活动后缀（active suffix）等概念，都无法达到线性时间复杂度的要求。

有一些 Ukkonen 算法的 PLT/Scheme 实现 [34]，在处理中不断更新后缀链结。这样的实现不是纯函数式的。

现有的纯函数式实现通常依赖惰性编程环境，如 Bryan O'Sullivan 使用 [32] 中的算法给出的 Haskell 实现 [33]。后缀树直到被查询或者遍历时才按需构建。但是在不支持惰性求值的环境中，这一算法不能保证 $O(n)$ 的性能。

下面的 Haskell 例子程序给出了后缀树的定义：一棵后缀树或者是一个叶子；或者是一棵包含多棵子树的分支，其中每棵子树都对应一个字符串。

```
data Tr = Lf | Br [(String, Tr)] deriving (Eq)
type EdgeFunc = [String]→(String, [String])
```

函数 edge 从一组字符串列表中提取出公共前缀。注意这里并不要求 edge 函数一定要返回最长的公共前缀，它也可以返回一个空串（空串显然是任何字符串的前缀）。我们可以使用不同的 edge 函数来产生出不同的树。

$$build(edge, X)$$

这是一个通用的基数树（radix）构造函数。它使用一个 edge 函数从一组字符串

中构造树。如果 X 是某个字符串的全部后缀，结果就是后缀 Trie 或者后缀树。如果 X 是某个字符串的全部前缀，结果将会是前缀 Trie 或者 Patricia。

设所有字符串含有的字符集为 Σ。若用于构造树的字符串为空，X 仅仅包含一个空串，构造的结果为一个空的叶子节点；否则，我们检查 Σ 中的每个字符，将 X 中的字符串按照起始字符分组，然后在每一组上应用 edge 函数。

$$build(edge, X) = \begin{cases} leaf & : X = \{\phi\} \\ branch(\{(\{c\} \cup p, \; build(edge, X')) | \\ \quad c \in \Sigma, \\ \quad G \in \{group(X, c)\}, \\ \quad (p, X') \in \{edge(G)\}\}) & : otherwise \end{cases} \tag{C.3}$$

算法首先将全部的后缀按照首字符分成若干组，然后将每组中的首字符移除。例如，后缀 { “acac”, “cac”, “ac”, “c” } 被分成两组：{(‘a’ , [“cac”, “c”]), (‘c’ , [“ac”, “”])}。

$$group(X, c) = \{C' | \{c_1\} \cup C' \in X, c_1 = c\} \tag{C.4}$$

函数 $group$ 枚举 X 中的全部后缀，每个后缀的首字符记为 c_1，余下字符记为 C'。如果 c_1 和传入的字符 c 相等，则相应的 C' 就被归并到一组。

下面的 Haskell 例子程序实现了通用的基数树构造算法：

```
alpha = ['a'..'z']++['A'..'Z']

lazyTree::EdgeFunc → [String] → Tr
lazyTree edge = build where
    build [[]] = Lf
    build ss = Br [(a:prefix, build ss') |
                        a←alpha,
                        xs@(x:_) ←[[cs | c:cs←ss, c==a]],
                        (prefix, ss')←[edge xs]]
```

我们前面提到，不同的 edge 函数会构造出不同的基数树。对 edge 函数的唯一的要求就是它必须能够提取出一组字符串的公共前缀。其中最简单的一种 edge 函数对任何输入都返回空串作为结果，这时我们会得到一棵 Trie。

$$edgeTrie(X) = (\phi, X) \tag{C.5}$$

我们也可以实现一个 edge 函数，它能够提取出最长的公共前缀。这样的 edge 函数会构造出 Patricia。记字符串列表为 $X = \{x_1, x_2, ..., x_n\}$，对每个字符串 x_i，令首字符为 c_i，剩余的字符为 W_i。若 X 仅含有一个字符串，最长的公共前缀显然就是这个字符串本身；如果 X 含有首字符不同的字符串，则公共前缀为空串；否则，所有的字符串的首字符都相同，这一首字符一定属于最长公共前缀。我们可以将所有字符

串的首字符去除，然后递归地调用 edge 函数来寻找最长公共前缀。

$$edgeTree(X) = \begin{cases} (x_1, \{\phi\}) & : & X = \{x_1\} \\ (\phi, X) & : & |X| > 1, \exists x_i \in X, c_i \neq c_1 \\ (\{c_1\} \cup p, Y) & : & (p, Y) = edgeTree(\{W_i | x_i \in X\}) \end{cases} \tag{C.6}$$

下面列出函数 $edgeTree$ 的一些例子。

$$edgeTree(\{\text{“an”, “another”, “and”}\}) = (\text{“an”}, \{\text{“”, “other”, “d”}\})$$
$$edgeTree(\{\text{“bool”, “foo, “bar”}\}) = (\text{“”}, \{\text{“bool”, “fool”, “bar”}\})$$

下面的 Haskell 例子程序实现了提取最长公共前缀的 edge 函数。

```
edgeTree::EdgeFunc
edgeTree [s] = (s, [[]])
edgeTree awss@((a:w):ss) | null [c|c:_←ss, a≠c] = (a:prefix, ss')
                         | otherwise             = ("", awss)
                         where (prefix, ss') = edgeTree (w:[u| _:u←ss])
edgeTree ss = ("", ss)
```

给定任意字符串，我们可以通过使用上面的两个 edge 函数来构造后缀 Trie 和后缀树。

$$suffixTrie(S) = build(edgeTrie, suffixes(S)) \tag{C.7}$$

$$suffixTree(S) = build(edgeTree, suffixes(S)) \tag{C.8}$$

由于 $build(edge, X)$ 的通用性，它也可以用于构造普通的前缀 Trie 和 Patricia 等基数树：

$$trie(S) = build(edgeTrie, prefixes(S)) \tag{C.9}$$

$$tree(S) = build(edgeTree, prefixes(S)) \tag{C.10}$$

C.4 后缀树的应用

后缀树可以高效地处理很多字符串操作和 DNA 匹配相关的问题。

C.4.1 字符串搜索和模式匹配

字符串搜索的算法非常丰富，例如著名的 KMP（Knuth-Morris-Pratt）算法，本书搜索一章专门介绍了这一算法。后缀树的搜索效率和 KMP 算法相当 [35]。如果待搜索的子串长度为 m，后缀树的搜索时间为 $O(m)$。但是我们需要 $O(n)$ 的时间来预先构造搜索树，其中 n 是搜索文本的长度 [36]。

不仅是字符串搜索，后缀树还可以用来实现模式匹配甚至正则表达式引擎。Ukkonen 将这类问题称为子串 motif。他指出：“即使字符串 S 可能含有 $O(n^2)$ 个子串。$SuffixTree(S)$ 也可以在 $O(n)$ 时间内找出任何子串 motif 的出现次数。”

子串出现的次数

$SuffixTree(S)$ 中，任意分支节点都代表 S 中出现一次以上的子串。如果子串在 S 中出现 k 次，则其对应的节点含有 k 个子分支 [37]。

```
1: function LOOKUP-PATTERN(T, s)
2:     loop
3:         match ← FALSE
4:         for ∀(s_i, T_i) ∈ VALUES(CHILDREN(T)) do
5:             if s ⊏ s_i then
6:                 return MAX(|CHILDREN(T_i)|, 1)
7:             else if s_i ⊏ s then
8:                 match ← TRUE
9:                 T ← T_i
10:                s ← s − s_i
11:                break
12:        if ¬match then
13:            return 0
```

当从文本 w 中寻找子串 s 时，我们首先从 w 构造后缀树 T。从根节点开始，我们遍历所有子节点。针对每对子串 s_i 和子树 T_i，检查 s 是否是 s_i 的前缀。如果是，就返回 T_i 的子分支个数作为结果。有一种特殊情况：T_i 是一个叶子节点，而没有任何子节点。这种情况下我们需要返回 1，而不是 0。因此上述实现中，我们使用了 max 函数。反之，如果 s_i 是 s 的前缀，我们就从 s 中去掉 s_i 这一部分，然后递归地在 T_i 中查找。

下面的 Python 例子程序实现了这一算法。

```python
def lookup_pattern(t, s):
    node = t.root
    while True:
        match = False
        for _, (str_ref, tr) in node.children.items():
            edge = substr(t, str_ref)
            if string.find(edge, s)==0: # s `isPrefixOf` edge
                return max(len(tr.children), 1)
            elif string.find(s, edge)==0: # edge `isPrefixOf` s
                match = True
                node = tr
                s = s[len(edge):]
                break
        if not match:
            return 0
    return 0 # not found
```

也可以用递归的方式查找子串出现的次数。若后缀树 T 不是一个简单的叶子节点，记 C 为 T 之下的全部“子串—子树”映射对：$C = \{(s_1, T_1), (s_2, T_2), ...\}$。我们

在这组子树中查找子串。

$$lookup_{pattern}(T, s) = find(C, s) \tag{C.11}$$

如果 C 为空，说明子串没有出现过；否则，我们检查第一对映射 (s_1, T_1)，如果 s 是 s_1 的前缀，则子树 T_1 的子分支数目就是结果；如果 s_1 是 s 的子串，我们从 s 中去除 s_1 的部分，然后递归在 T_1 中查找；否则，我们继续在剩余的“子串—子树”映射 C' 中进行同样的处理。

$$find(C, s) = \begin{cases} 0 & : C = \phi \\ max(1, |C_1|) & : s \sqsubset s_1 \\ lookup_{pattern}(T_1, s - s_1) & : s_1 \sqsubset s \\ find(C', s) & : otherwise \end{cases} \tag{C.12}$$

下面的 Haskell 例子程序实现了这一算法。

```
lookupPattern (Br lst) ptn = find lst where
    find [] = 0
    find ((s, t):xs)
         | ptn `isPrefixOf` s = numberOfBranch t
         | s `isPrefixOf` ptn = lookupPattern t (drop (length s) ptn)
         | otherwise = find xs
    numberOfBranch (Br ys) = length ys
    numberOfBranch _ = 1

findPattern s ptn = lookupPattern (suffixTree $ s++"$") ptn
```

我们总是在字符串末尾增加一个特殊的终结符（上述程序中使用‘$’作为终结符），这样就可以避免某个后缀也同时也是其它后缀的前缀 [23]。

后缀树也可以用来搜索“a**n”这样的模式，本书略过了这些内容，读者可以参考 [37] 或 [38] 来了解其中的细节。

C.4.2 查找最长重复子串

向字符串 S 增加一个特殊的终结符，我们可以通过在后缀树中搜索最深分支来找到最长重复子串。

考虑图C.11中的后缀树。

深度为 3 的分支节点有 3 个，分别是 A、B 和 C。其中 A 代表了最长重复子串“issi”，B 代表“si”，C 代表“ssi”，它们都比 A 代表的子串短。

这个例子说明，分支的“深度”应该用从根节点开始到达分支所经过的字符数目来衡量，而不是使用可见分支节点的个数来衡量。

下面的广度优先搜索算法，可以在后缀树中找到最长重复子串。

1: **function** LONGEST-REPEATED-SUBSTRING(T)
2: $Q \leftarrow$ (NIL, ROOT(T))
3: $R \leftarrow$ NIL

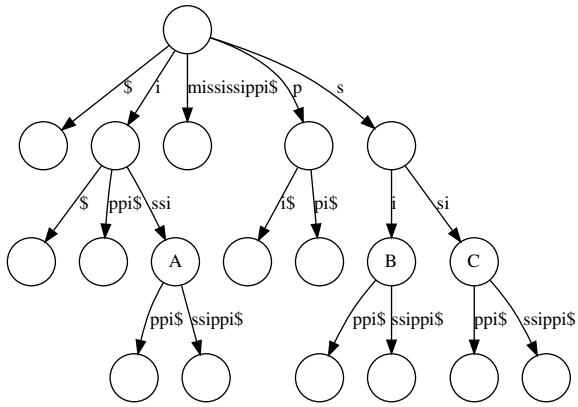


图 C.11: 字符串‘mississippi$’对应的后缀树

```
4:    while Q is not empty do
5:        (s, T) ← POP(Q)
6:        for each ((l, r), T') ∈ CHILDREN(T) do
7:            if T' is not leaf then
8:                s' ← CONCATENATE(s, (l, r))
9:                PUSH(Q, (s', T'))
10:               R ← UPDATE(R, s')
11:   return R
```

算法使用一个队列来进行搜索，队列一开始含有一对元素，包括一个空串和根节点。然后不断从队列中取出候选元素进行处理。

针对每个节点，我们都逐一检查它的所有子树。如果是一个分支节点，这个子树就被放回队列等待后继的搜索。它对应的子串就被作为最长重复子串的一个候选。

函数 $\textsc{Update}(R, s')$ 用于更新候选的最长重复子串。如果存在多个长度相同的子串，它们会被放入一个结果列表中。

```
1: function UPDATE(L, s)
2:     if L = NIL ∨|l_1| < |s| then
3:         return l ← {s}
4:     if |l_1| = |s| then
5:         return APPEND(L, s)
6:     return L
```

下面的 Python 例子程序实现了这一算法。

```
def lrs(t):
    queue = [("", t.root)]
    res = []
    while len(queue)>0:
        (s, node) = queue.pop(0)
```

```
        for _, (str_ref, tr) in node.children.items():
            if len(tr.children)>0:
                s1 = s+t.substr(str_ref)
                queue.append((s1, tr))
                res = update_max(res, s1)
    return res

def update_max(lst, x):
    if lst ==[] or len(lst[0]) < len(x):
        return [x]
    if len(lst[0]) == len(x):
        return lst + [x]
    return lst
```

我们也可以用递归的方式搜索最长重复子串。如果待搜索的树是一个叶子节点，结果为空串；否则算法在子树中寻找最长重复子串。

$$LRS(T) = \begin{cases} \phi & : & leaf(T) \\ longest(\{s_i \cup LRS(T_i) | (s_i, T_i) \in C, \neg leaf(T_i)\}) & : & otherwise \end{cases} \tag{C.13}$$

下面的 Haskell 例子程序实现了最长重复子串的算法。

```
isLeaf Lf = True
isLeaf _ = False

lrs' Lf = ""
lrs' (Br lst) = find $ filter (not ∘ isLeaf ∘ snd) lst where
    find [] = ""
    find ((s, t):xs) = maximumBy (compare `on` length) [s++(lrs' t), find xs]
```

C.4.3 查找最长公共子串

后缀树还可以用来查找多个字符串的最长公共子串。我们先考虑两个字符串的情况。记这两个串为 txt_1 和 txt_2，我们可以构造一棵后缀树 $SuffixTree(txt_1\$_1 txt_2\$_2)$。其中 $\$_1$ 是 txt_1 的特殊终结符；$\$_2 \neq \$_1$ 是 txt_2 的特殊终结符。

为了获得最长公共子串，我们只需要找到最深的一个分支节点，它同时包含形如“...$\$_1$...”和“...$\$_2$”（不含 $\$_1$）的两个叶子。这里“最深”节点的含意和前面最长重复子串中的一致：深度等于从根节点算起的字符个数。

如果一个节点含有代表“...$\$_1$...”的叶子，这个节点一定对应 txt_1 的某个子串。同时，由于它也含有一个代表“...$\$_2$”（不含 $\$_1$）的叶子，这个节点一定也对应到 txt_2 的某个子串。如果它是所有这类节点中最深的一个，它一定对应着最长的公共子串。

我们可以使用类似的广度优先（BFS）算法来查找最长公共子串。

1: **function** LONGEST-COMMON-SUBSTRING(T)
2: $Q \leftarrow$ (NIL, ROOT(T))
3: $R \leftarrow$ NIL

```
    while Q is not empty do
        (s, T) ← POP(Q)
        if MATCH-FORK(T) then
            R ← UPDATE(R, s)
        for each ((l, r), T′) ∈ CHILDREN(T) do
            if T′ is not leaf then
                s′ ← CONCATENATE(s, (l, r))
                PUSH(Q, (s′, T′))
    return R
```

大部分实现和最长重复子串的查找算法相同。函数 MATCH-FORK 用以检查一个节点是否含有两个满足公共子串的叶子节点。

```
function MATCH-FORK(T)
    if | CHILDREN(T) | = 2 then
        {(s_1, T_1), (s_2, T_2)} ← CHILDREN(T)
        return T_1 is leaf ∧T_2 is leaf ∧ XOR($_1 ∈ s_1, $_1 ∈ s_2))
    return FALSE
```

这个函数检查一个节点是否含有两个叶子节点，其中一个含有 $\$_2$，而另外一个不含有。这是因为如果一个节点是叶子节点，根据后缀树的定义，它一定包含终结符 $\$_1$。

下面的 Python 例子程序实现了最长公共子串的查找算法。

```python
def lcs(t):
    queue = [("", t.root)]
    res = []
    while len(queue)>0:
        (s, node) = queue.pop(0)
        if match_fork(t, node):
            res = update_max(res, s)
        for _, (str_ref, tr) in node.children.items():
            if len(tr.children)>0:
                s1 = s + t.substr(str_ref)
                queue.append((s1, tr))
    return res

def is_leaf(node):
    return node.children=={}

def match_fork(t, node):
    if len(node.children)==2:
        [(_, (str_ref1, tr1)), (_, (str_ref2, tr2))]=node.children.items()
        return is_leaf(tr1) and is_leaf(tr2) and
            (t.substr(str_ref1).find('#')≠-1) ≠
            (t.substr(str_ref2).find('#')≠-1)
    return False
```

我们也可以用递归的方式实现最长公共子串的查找算法。如果后缀树 T 是一个叶子节点，则查找失败，两个字符串间不存在公共子串；否则，我们逐一检查树的全部子分支，记录下所有满足公共子串条件的候选节点；并且递归在不满足条件的子树中查找。最后我们将最长的候选子串返回。

$$LCS(T) = \begin{cases} \phi & : leaf(T) \\ longest(\{s_i | (s_i, T_i) \in C, match(T_i)\} \cup \{s_i \cup LCS(T_i) | (s_i, T_i) \in C, \neg match(T_i)\}) & : otherwise \end{cases} \tag{C.14}$$

下面的 Haskell 例子程序实现了最长公共子串的查找算法。

```
lcs Lf = []
lcs (Br lst) = find $ filter (not ∘ isLeaf ∘ snd) lst where
    find [] = []
    find ((s, t):xs) = maxBy (compare `on` length)
                       (if match t
                        then s:(find xs)
                        else  (map (s++) (lcs t)) ++ (find xs))

match (Br [(s1, Lf), (s2, Lf)]) = ("#" `isInfixOf` s1) ≠ ("#" `isInfixOf` s2)
match _ = False
```

C.4.4 查找最长回文

回文是一种特殊的字符串 S，满足 $S = reverse(S)$。例如“level”、“rotator”和“civic”都是回文。

给定字符串 $s_1 s_2 ... s_n$，利用后缀树，我们可以在线性时间 $O(n)$ 内找到它包含的最长回文。我们可以在最长公共子串查找算法的基础上解决回文问题。

如果字符串 S 的某个子串 w 是一个回文，则 w 一定也是 $reverse(S)$ 的子串。例如“issi”是“mississippi”的子串，同时它也是反转的字符串“ippississim”的子串。

根据这一点，我们可以通过寻找 S 和 $reverse(S)$ 的最长公共子串来获得最长回文。

$$palindrome_m(S) = LCS(suffixTree(S \cup reverse(S))) \tag{C.15}$$

下面的 Haskell 例子程序实现了最长回文的查找算法。

```
longestPalindromes s = lcs $ suffixTree (s++"#"++(reverse s)++"$")
```

但是仅仅这样还不够，例如字符串“abacoxyocaba”和它的反转串“abacoyxocaba”的最长公共子串是“abaco”，但它却不是一个回文。这是因为“abaco”的反转结果“ocaba”也包含在原字符串中。为了排除这种情况，我们需要进一步判断，在 S 和 $reverse(S)$ 中找出的公共子串，是否来自 S 中的同一位置。具体说，若公共子串 w

的长度为 L，S 的长度为 n，w 在 S 中的位置为 i，在 $reverse(S)$ 中的位置为 j，我们有：

$$j = n - i + L \tag{C.16}$$

为了加入这一限制条件，我们可以改动最长公共子串查找算法。当找到一个中间节点，它同时包含形如“...$\$_1$...”和“...$\$_2$”（不含 $\$_1$）的两个叶子时，我们进一步检查上述位置限制条件。只有他们来自 S 中的相同位置，才会被作为最长回文的候选。

C.4.5 其它

后缀树还可以用于数据压缩，例如 Burrows-Wheeler 变换，LZW 压缩（LZSS）等 [23]。

C.5 小结

后缀树最早由 Weiner 在 1973 年引入 [28]。McCreight 在 1976 年大幅简化了后缀树的构造算法。他的方法从右向左构造后缀树。1995 年，Ukkonen 给出了第一个从左向右的 on-line 构造算法。这三种方法都是线性时间算法（$O(n)$ 时间），近来的研究发现，它们彼此之间有着紧密的联系 [31]。

参考文献

[1] Richard Bird. “Pearls of functional algorithm design”. Cambridge University Press; 1 edition (November 1, 2010). ISBN-10: 0521513383. pp1 - pp6.

[2] Jon Bentley. “Programming Pearls(2nd Edition)”. Addison-Wesley Professional; 2 edition (October 7, 1999). ISBN-13: 978-0201657883 （中文版:《编程珠玑》）

[3] Chris Okasaki. “Purely Functional Data Structures”. Cambridge university press, (July 1, 1999), ISBN-13: 978-0521663502

[4] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein. “Introduction to Algorithms, Second Edition”. The MIT Press, 2001. ISBN: 0262032937. （中文版:《算法导论》）

[5] Chris Okasaki. “Ten Years of Purely Functional Data Structures”. http://okasaki.blogspot.com/2008/02/ten-years-of-purely-functional-data.html

[6] SGI. “Standard Template Library Programmer’s Guide”. http://www.sgi.com/tech/stl/

[7] Wikipedia. “Fold(high-order function)”. https://en.wikipedia.org/wiki/Fold_(higher-order_function)

[8] Wikipedia. “Function Composition”. http://en.wikipedia.org/wiki/Function_composition

[9] Wikipedia. “Partial application”. http://en.wikipedia.org/wiki/Partial_application

[10] Miran Lipovaca. “Learn You a Haskell for Great Good! A Beginner’s Guide”. No Starch Press; 1 edition April 2011, 400 pp. ISBN: 978-1-59327-283-8

[11] Wikipedia. “Bubble sort”. http://en.wikipedia.org/wiki/Bubble_sort

[12] Donald E. Knuth. “The Art of Computer Programming, Volume 3: Sorting and Searching (2nd Edition)”. Addison-Wesley Professional; 2 edition (May 4, 1998) ISBN-10: 0201896850 ISBN-13: 978-0201896855

[13] Chris Okasaki. “FUNCTIONAL PEARLS Red-Black Trees in a Functional Setting”. J. Functional Programming. 1998

[14] Wikipedia. “Red-black tree”. http://en.wikipedia.org/wiki/Red-black_tree

[15] Lyn Turbak. “Red-Black Trees”. http://cs.wellesley.edu/~cs231/fall01/red-black.pdf Nov. 2, 2001.

[16] Rosetta Code. “Pattern matching”. http://rosettacode.org/wiki/Pattern_matching

[17] Hackage. “Data.Tree.AVL”. http://hackage.haskell.org/packages/archive/AvlTree/4.2/doc/html/Data-Tree-AVL.html

[18] Wikipedia. “AVL tree”. http://en.wikipedia.org/wiki/AVL_tree

[19] Guy Cousinear, Michel Mauny. “The Functional Approach to Programming”. Cambridge University Press; English Ed edition (October 29, 1998). ISBN-13: 978-0521576819

[20] Pavel Grafov. “Implementation of an AVL tree in Python”. http://github.com/pgrafov/python-avl-tree

[21] Chris Okasaki and Andrew Gill. “Fast Mergeable Integer Maps”. Workshop on ML, September 1998, pages 77-86.

[22] D.R. Morrison, “PATRICIA – Practical Algorithm To Retrieve Information Coded In Alphanumeric”, Journal of the ACM, 15(4), October 1968, pages 514-534.

[23] Wikipedia. “Suffix Tree”. http://en.wikipedia.org/wiki/Suffix_tree

[24] Wikipedia. “Trie”. http://en.wikipedia.org/wiki/Trie

[25] Wikipedia. “T9 (predictive text)”. http://en.wikipedia.org/wiki/T9_(predictive_text)

[26] Wikipedia. “Predictive text”. http://en.wikipedia.org/wiki/Predictive_text

[27] Esko Ukkonen. “On-line construction of suffix trees”. Algorithmica 14 (3): 249–260. doi:10.1007/BF01206331. http://www.cs.helsinki.fi/u/ukkonen/SuffixT1withFigs.pdf

[28] Weiner, P. “Linear pattern matching algorithms”, 14th Annual IEEE Symposium on Switching and Automata Theory, pp. 1-11, doi:10.1109/SWAT.1973.13

[29] Esko Ukkonen. “Suffix tree and suffix array techniques for pattern analysis in strings”. http://www.cs.helsinki.fi/u/ukkonen/Erice2005.ppt

[30] Suffix Tree (Java). http://en.literateprograms.org/Suffix_tree_(Java)

[31] Robert Giegerich and Stefan Kurtz. “From Ukkonen to McCreight and Weiner: A Unifying View of Linear-Time Suffix Tree Construction”. Science of Computer Programming 25(2-3):187-218, 1995. http://citeseer.ist.psu.edu/giegerich95comparison.html

[32] Robert Giegerich and Stefan Kurtz. “A Comparison of Imperative and Purely Functional Suffix Tree Constructions”. Algorithmica 19 (3): 331–353. doi:10.1007/PL00009177. www.zbh.uni-hamburg.de/pubs/pdf/GieKur1997.pdf

[33] Bryan O’Sullivan. “suffixtree: Efficient, lazy suffix tree implementation”. http://hackage.haskell.org/package/suffixtree

[34] Danny. http://hkn.eecs.berkeley.edu/~dyoo/plt/suffixtree/

[35] Dan Gusfield. “Algorithms on Strings, Trees and Sequences Computer Science and Computational Biology”. Cambridge University Press; 1 edition (May 28, 1997) ISBN: 9780521585194

[36] Lloyd Allison. “Suffix Trees”. http://www.allisons.org/ll/AlgDS/Tree/Suffix/

[37] Esko Ukkonen. “Suffix tree and suffix array techniques for pattern analysis in strings”. http://www.cs.helsinki.fi/u/ukkonen/Erice2005.ppt

[38] Esko Ukkonen “Approximate string-matching over suffix trees”. Proc. CPM 93. Lecture Notes in Computer Science 684, pp. 228-242, Springer 1993. http://www.cs.helsinki.fi/u/ukkonen/cpm931.ps

[39] Wikipeida. “B-tree”. http://en.wikipedia.org/wiki/B-tree

[40] Wikipedia. “Heap (data structure)”. http://en.wikipedia.org/wiki/Heap_(data_structure)

[41] Wikipedia. “Heapsort”. http://en.wikipedia.org/wiki/Heapsort

[42] Rosetta Code. “Sorting algorithms/Heapsort”. http://rosettacode.org/wiki/Sorting_algorithms/Heapsort

[43] Wikipedia. “Leftist Tree”. http://en.wikipedia.org/wiki/Leftist_tree

[44] Bruno R. Preiss. Data Structures and Algorithms with Object-Oriented Design Patterns in Java. http://www.brpreiss.com/books/opus5/index.html

[45] Donald E. Knuth. “The Art of Computer Programming. Volume 3: Sorting and Searching.”. Addison-Wesley Professional; 2nd Edition (October 15, 1998). ISBN-13: 978-0201485417. Section 5.2.3 and 6.2.3

[46] Wikipedia. “Skew heap”. http://en.wikipedia.org/wiki/Skew_heap

[47] Sleator, Daniel Dominic; Jarjan, Robert Endre. “Self-adjusting heaps” SIAM Journal on Computing 15(1):52-69. doi:10.1137/0215004 ISSN 00975397 (1986)

[48] Wikipedia. “Splay tree”. http://en.wikipedia.org/wiki/Splay_tree

[49] Sleator, Daniel D.; Tarjan, Robert E. (1985), “Self-Adjusting Binary Search Trees”, Journal of the ACM 32(3):652 - 686, doi: 10.1145/3828.3835

[50] NIST, “binary heap”. http://xw2k.nist.gov/dads//HTML/binaryheap.html

[51] Donald E. Knuth. “The Art of Computer Programming, Volume 3: Sorting and Searching (2nd Edition)”. Addison-Wesley Professional; 2 edition (May 4, 1998) ISBN-10: 0201896850 ISBN-13: 978-0201896855

[52] Wikipedia. “Strict weak order”. http://en.wikipedia.org/wiki/Strict_weak_order

[53] Wikipedia. “FIFA world cup”. http://en.wikipedia.org/wiki/FIFA_World_Cup

[54] Wikipedia. “K-ary tree”. http://en.wikipedia.org/wiki/K-ary_tree

[55] Wikipedia, “Pascal’s triangle”. http://en.wikipedia.org/wiki/Pascal's_triangle

[56] Hackage. “An alternate implementation of a priority queue based on a Fibonacci heap.”, http://hackage.haskell.org/packages/archive/pqueue-mtl/1.0.7/doc/html/src/Data-Queue-FibQueue.html

[57] Chris Okasaki. “Fibonacci Heaps.” http://darcs.haskell.org/nofib/gc/fibheaps/orig

[58] Michael L. Fredman, Robert Sedgewick, Daniel D. Sleator, and Robert E. Tarjan. “The Pairing Heap: A New Form of Self-Adjusting Heap” Algorithmica (1986) 1: 111-129.

[59] Maged M. Michael and Michael L. Scott. “Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms”. http://www.cs.rochester.edu/research/synchronization/pseudocode/queues.html

[60] Herb Sutter. “Writing a Generalized Concurrent Queue”. Dr. Dobb’s Oct 29, 2008. http://drdobbs.com/cpp/211601363?pgno=1

[61] Wikipedia. “Tail-call”. http://en.wikipedia.org/wiki/Tail_call

[62] Wikipedia. “Recursion (computer science)”. http://en.wikipedia.org/wiki/Recursion_(computer_science)#Tail-recursive_functions

[63] Harold Abelson, Gerald Jay Sussman, Julie Sussman. “Structure and Interpretation of Computer Programs, 2nd Edition”. MIT Press, 1996, ISBN 0-262-51087-1 (中文版：裘宗燕译《计算机程序的构造和解释》)

[64] Chris Okasaki. “Purely Functional Random-Access Lists”. Functional Programming Languages and Computer Architecture, June 1995, pages 86-95.

[65] Ralf Hinze and Ross Paterson. “Finger Trees: A Simple General-purpose Data Structure.” in Journal of Functional Programming16:2 (2006), pages 197-217. http://www.soi.city.ac.uk/~ross/papers/FingerTree.html

[66] Guibas, L. J., McCreight, E. M., Plass, M. F., Roberts, J. R. (1977), ”A new representation for linear lists”. Conference Record of the Ninth Annual ACM Symposium on Theory of Computing, pp. 49-60.

[67] Generic finger-tree structure. http://hackage.haskell.org/packages/archive/fingertree/0.0/doc/html/Data-FingerTree.html

[68] Wikipedia. “Move-to-front transform”. http://en.wikipedia.org/wiki/Move-to-front_transform

[69] Robert Sedgewick. “Implementing quick sort programs”. Communication of ACM. Volume 21, Number 10. 1978. pp.847 - 857.

[70] Jon Bentley, Douglas McIlroy. “Engineering a sort function”. Software Practice and experience VOL. 23(11), 1249-1265 1993.

[71] Robert Sedgewick, Jon Bentley. “Quicksort is optimal”. http://www.cs.princeton.edu/~rs/talks/QuicksortIsOptimal.pdf

[72] Fethi Rabhi, Guy Lapalme. “Algorithms: a functional programming approach”. Second edition. Addison-Wesley, 1999. ISBN: 0201-59604-0

[73] Simon Peyton Jones. “The Implementation of functional programming languages”. Prentice-Hall International, 1987. ISBN: 0-13-453333-X

[74] Jyrki Katajainen, Tomi Pasanen, Jukka Teuhola. “Practical in-place mergesort”. Nordic Journal of Computing, 1996.

[75] Josè Bacelar Almeida and Jorge Sousa Pinto. “Deriving Sorting Algorithms”. Technical report, Data structures and Algorithms. 2008.

[76] Cole, Richard (August 1988). “Parallel merge sort”. SIAM J. Comput. 17 (4): 770-785. doi:10.1137/0217049. (August 1988)

[77] Powers, David M. W. “Parallelized Quicksort and Radixsort with Optimal Speedup”, Proceedings of International Conference on Parallel Computing Technologies. Novosibirsk. 1991.

[78] Wikipedia. “Quicksort”. http://en.wikipedia.org/wiki/Quicksort

[79] Wikipedia. “Total order”. http://en.wokipedia.org/wiki/Total_order

[80] Wikipedia. “Harmonic series (mathematics)”. http://en.wikipedia.org/wiki/Harmonic_series_(mathematics)

[81] M. Blum, R.W. Floyd, V. Pratt, R. Rivest and R. Tarjan, ”Time bounds for selection,” J. Comput. System Sci. 7 (1973) 448-461.

[82] Edsger W. Dijkstra. “The saddleback search”. EWD-934. 1985. http://www.cs.utexas.edu/users/EWD/index09xx.html.

[83] Robert Boyer, and Strother Moore. “MJRTY - A Fast Majority Vote Algorithm”. Automated Reasoning: Essays in Honor of Woody Bledsoe, Automated Reasoning Series, Kluwer Academic Publishers, Dordrecht, The Netherlands, 1991, pp. 105-117.

[84] Cormode, Graham; S. Muthukrishnan (2004). “An Improved Data Stream Summary: The Count-Min Sketch and its Applications”. J. Algorithms 55: 29-38.

[85] Knuth Donald, Morris James H., jr, Pratt Vaughan. “Fast pattern matching in strings”. SIAM Journal on Computing 6 (2): 323-350. 1977.

[86] Robert Boyer, Strother Moore. “A Fast String Searching Algorithm”. Comm. ACM (New York, NY, USA: Association for Computing Machinery) 20 (10): 762-772. 1977

[87] R. N. Horspool. “Practical fast searching in strings”. Software - Practice & Experience 10 (6): 501-506. 1980.

[88] Wikipedia. “Boyer-Moore string search algorithm”. http://en.wikipedia.org/wiki/Boyer-Moore_string_search_algorithm

[89] Wikipedia. “Eight queens puzzle”. http://en.wikipedia.org/wiki/Eight_queens_puzzle

[90] George Pólya. “How to solve it: A new aspect of mathematical method”. Princeton University Press(April 25, 2004). ISBN-13: 978-0691119663

[91] Wikipedia. “David A. Huffman”. http://en.wikipedia.org/wiki/David_A._Huffman

[92] Andrei Alexandrescu. “Modern C++ design: Generic Programming and Design Patterns Applied”. Addison Wesley February 01, 2001, ISBN 0-201-70431-5

[93] Benjamin C. Pierce. “Types and Programming Languages”. The MIT Press, 2002. ISBN:0262162091

[94] Joe Armstrong. “Programming Erlang: Software for a Concurrent World”. Pragmatic Bookshelf; 1 edition (July 18, 2007). ISBN-13: 978-1934356005

[95] SGI. “transform”. http://www.sgi.com/tech/stl/transform.html

[96] ACM/ICPC. “The drunk jailer.” Peking University judge online for ACM/ICPC. http://poj.org/problem?id=1218.

[97] Haskell wiki. “Haskell programming tips”. 4.4 Choose the appropriate fold. http://www.haskell.org/haskellwiki/Haskell_programming_tips

[98] Wikipedia. “Dot product”. http://en.wikipedia.org/wiki/Dot_product

[99] Xinyu LIU. “Isomorphism - mathematics of programming”. https://github.com/liuxinyu95/unplugged

索引