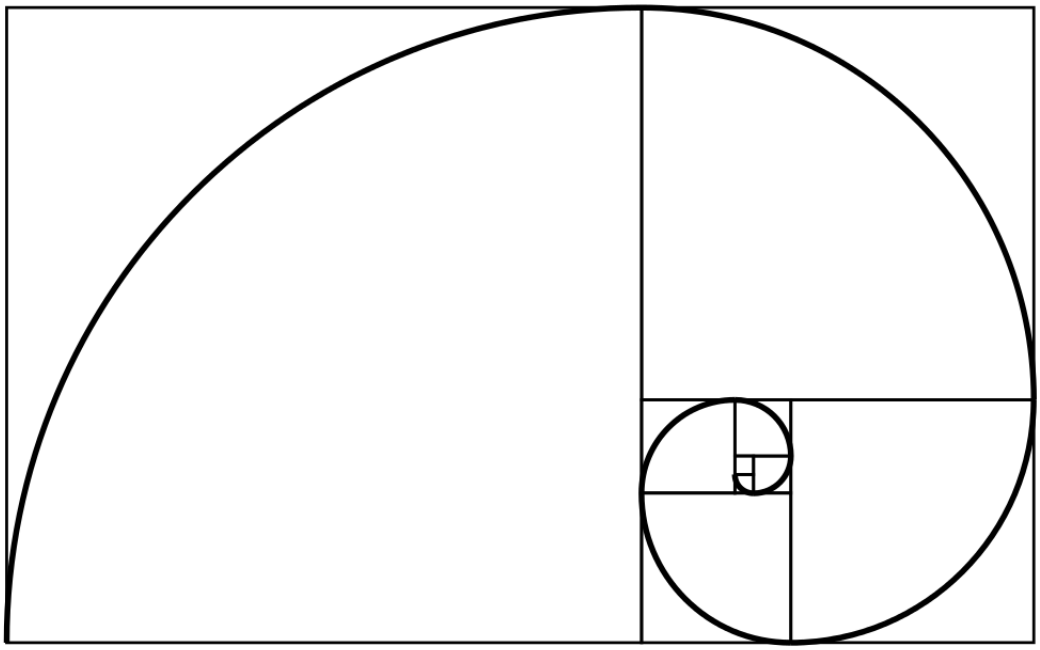
基本算法



刘新宇 [1]

2022 年 12 月 23 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: $x \leftarrow 0$
3: **loop**
4: **if** $x \notin A$ **then**
5: **return** x
6: **else**
7: $x \leftarrow x + 1$

其中符号 $\notin$ 的实现如下:

1: **function** '$\notin$'(x, X)
2: **for** $i \leftarrow 1$ 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。

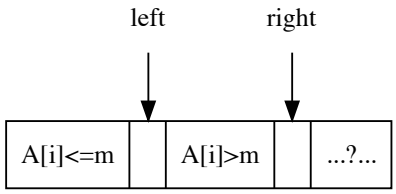


图 1: 数组划分。位于 $0 \le i < left$ 的元素满足 $A[i] \le m$, 位于 $left \le i < right$ 的元素满足 $A[i] > m$, 剩余的元素尚未处理。

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:

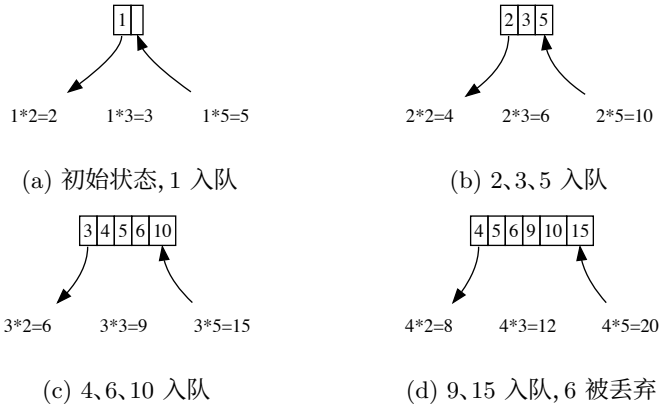
```
1: function REGULAR-NUMBER(n)
2:     x ← 1
3:     while n > 0 do
4:         x ← x + 1
5:         if VALID?(x) then
6:             n ← n − 1
7:     return x

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。

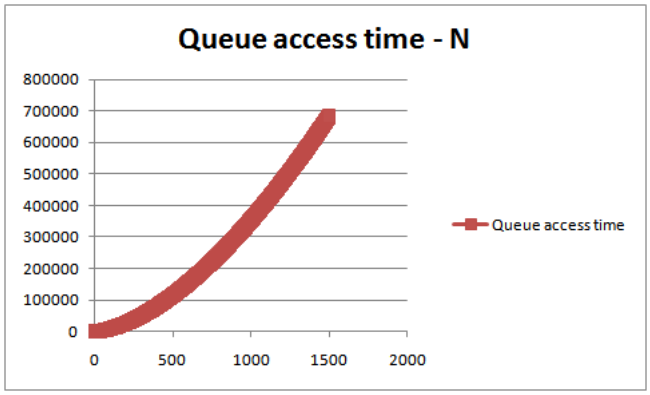


图 3: 队列访问次数和 n 的关系

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。

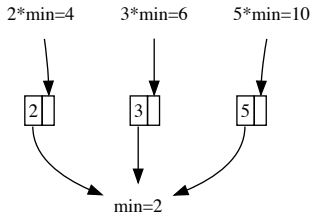
0.2.3 队列

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：

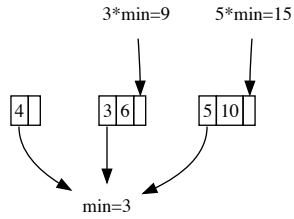
- 如果 x 是从 Q_2 取出的，我们将 $2x$ 加入 Q_2，$3x$ 加入 Q_{23}，$5x$ 加入 Q_{235}。
- 如果 x 是从 Q_{23} 取出的，我们只将 $3x$ 加入 Q_{23}，$5x$ 加入 Q_{235}。我们不应该将 $2x$ 加入 Q_2，因为 Q_2 中不允许包含被 3 整除的数。
- 如果 x 是从 Q_{235} 取出的，我们只将 $5x$ 加入 Q_{235}。我们不应该将 $2x$ 加入 Q_2，$3x$ 加入 Q_{23}，因为它们不允许包含被 5 整除的数。

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。

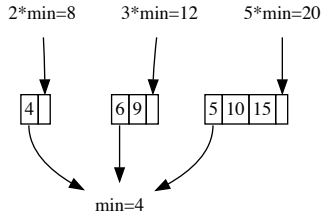
1: **function** REGULAR-NUMBER(n)
2: $x \leftarrow 1$
3: $Q_2 \leftarrow \{2\}$, $Q_{23} \leftarrow \{3\}$, $Q_{235} \leftarrow \{5\}$
4: **while** $n > 0$ **do**
5: $x \leftarrow min(\text{HEAD}(Q_2), \text{HEAD}(Q_{23}), \text{HEAD}(Q_{235}))$



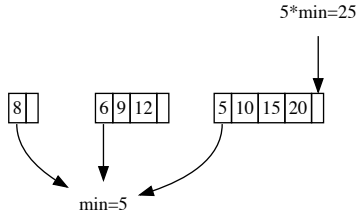
(a) 4、6、10 入队



(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, 找到第一个正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'。

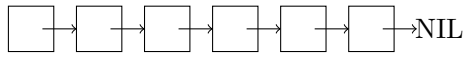


图 1.1: 由节点组成的列表

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')
```

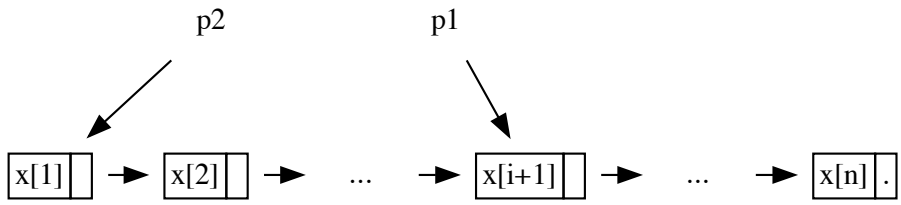
这一算法一边向尾部前进, 一边通过 cons 累积 init 的结果。但是这样产生的列表是逆序的, 因此最后需要将结果倒转过来(见第1.3.2节)。

1.2.5 反向索引

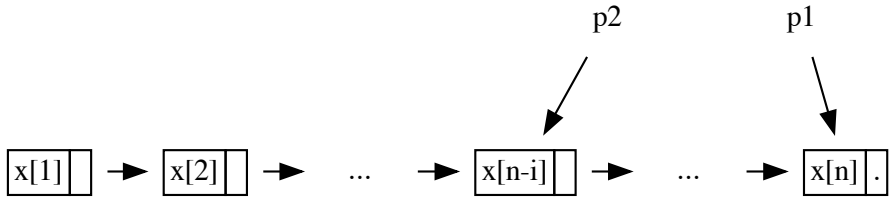
$last()$ 是反向索引的一种特例。更一般的形式是获取列表中的倒数第 i 个元素。最直接的思路是遍历两次: 第一次获取列表长度 n, 第二次获取第 $n - i - 1$ 个元素:

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

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 X 的表头 x_1;
- 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'。

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

其中函数 $slide(X, Y)$ 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：

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

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。

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

- 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:

```
function CONCAT(X, Y)
    if X = NIL then
        return Y
    if Y = NIL then
        return X
    H ← X
    while REST(X) ≠ NIL do
        X ← REST(X)
    REST(X) ← Y
    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：

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

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:

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

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**。

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

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 $P_n^r = \frac{n!}{(n-r)!}$ 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,
 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。

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

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)
2:     if L = NIL then
3:         return [NIL]
4:     x ← FIRST(L)
5:     L ← REST(L)
6:     g ← [x]
7:     G ← [g]
8:     while L ≠ NIL do
9:         y ← FIRST(L)
10:        if x ∼ y then
11:            g ← APPEND(g, y)
```

```
12:         else
13:             g ← [y]
14:             G ← Append(G, g)
15:         x ← y
16:         L ← Next(L)
17:     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(≥)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]。

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 $\varnothing$;

[5] 也叫作 reduce

- 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。

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

$$\begin{aligned} product([1,2,3,4,5]) &= 1 \times (2 \times (3 \times (4 \times 5))) \\ &= 1 \times (2 \times (3 \times 20)) \\ &= 1 \times (2 \times 60) \\ &= 1 \times 120 \\ &= 120 \end{aligned}$$

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:

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

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)

```
    if A = NIL then
        return TRUE
    n ← |A|
    while B ≠ NIL and n ≤ |B| do
        if A ⊆ B then
            return TRUE
        B ← REST(B)
    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: $C \leftarrow$ NIL
3: **while** $A \neq$ NIL and $B \neq$ NIL **do**
4: $C \leftarrow$ APPEND(C, (FIRST(A), FIRST(B))) ▷ 线性时间
5: $A \leftarrow$ REST(A)
6: $B \leftarrow$ 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)。

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

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

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。
5. 使用 zip 实现 $lastAt$。

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, $n = d_m...d_2d_1$ 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）。

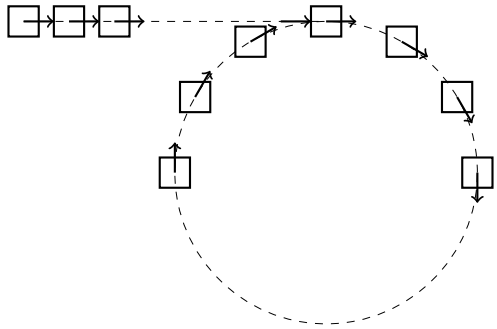


图 1.5: 带有循环的列表

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
```

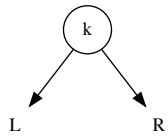
这里的 map 是用二叉搜索树实现的字典数据结构。我们用单词作为 key, 用单词出现的次数作为值。这个程序运行快速, 展示了二叉搜索树的强大功能。在详细介绍之前, 我们先来了解一下二叉树。

2.1　定义

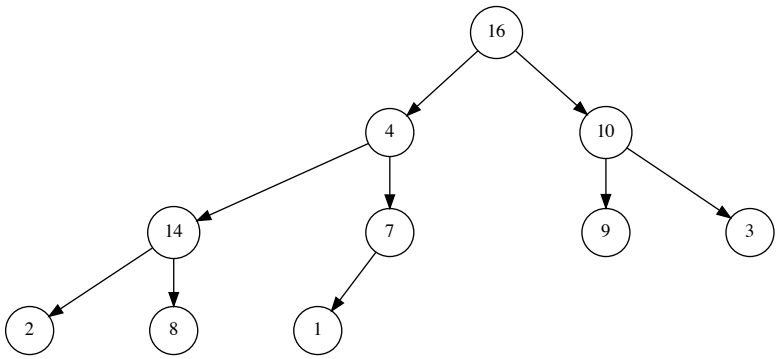
二叉树是一种递归的数据结构, 一棵二叉树:

- 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)。

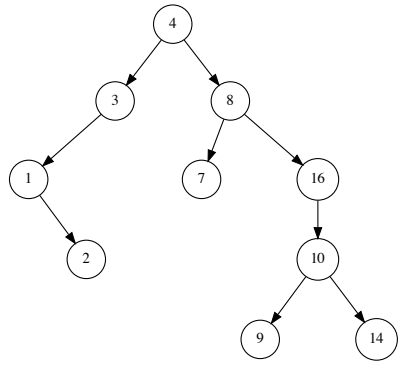


图 2.2: 二叉搜索树的例子

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:

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

[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。

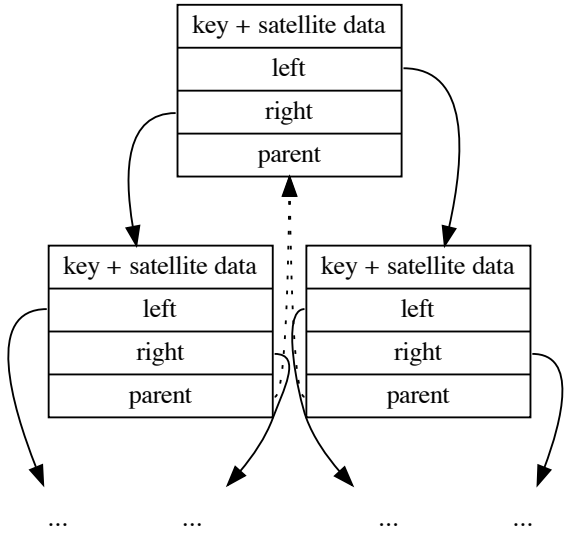


图 2.3: 带有父节点引用的数据组织

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。

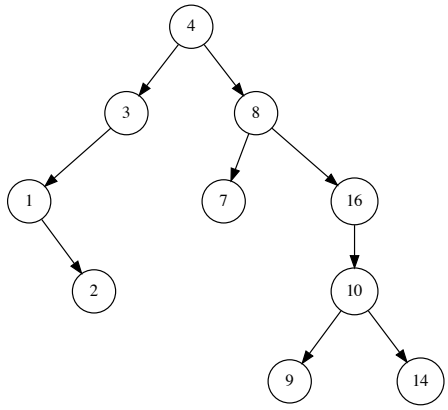


图 2.4: 8 的后继为其右侧分支的最小值 9; 为了获得 2 的后继, 首先向上找到 1, 它没有左子树, 所以继续向上找到 3, 3 的左子树也是 2 的祖先, 故而后继为 3。

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

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

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”。

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。

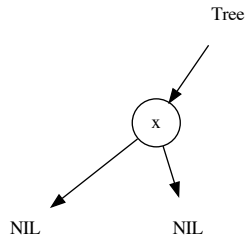


图 2.5: 叶子节点 x 可以直接“切下”

根据这个思路, 我们定义下面的 $delete$ 函数:

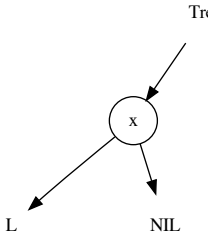
$$\begin{aligned} delete(\varnothing, x) &= \varnothing \\ delete(Node(T_l, k, T_r), x) &= \begin{cases} x < k: & Node(delete(T_l, x), k, T_r) \\ x > k: & Node(T_l, k, delete(T_r, x)) \\ x = k: & del(T_l, T_r) \end{cases} \end{aligned} \tag{2.9}$$

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。

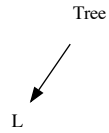
$$\begin{aligned} del(\varnothing, T_r) &= T_r \\ del(T_l, \varnothing) &= T_l \\ del(T_l, T_r) &= Node(T_l, y, delete(T_r, y)) \end{aligned} \tag{2.10}$$

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:

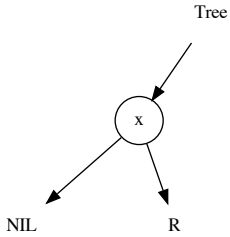
```
delete Empty _ = Empty
delete (Node l k r) x | x < k = Node (delete l x) k r
```



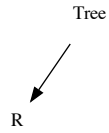
(a) 删除 x 前



(b) 删除 x 后。x 被“切掉”并由其左侧分支代替

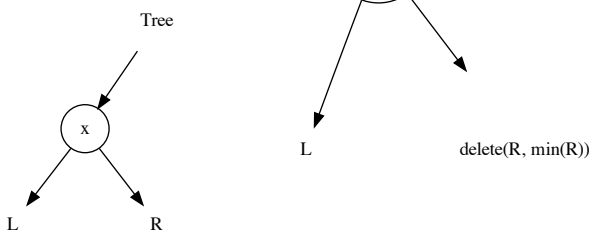


(c) 删除 x 前



(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

```
                    | 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 映射数据结构

我们可以用二叉搜索树来实现映射数据结构（Map，也称为关联数据结构或字典）。一个有限映射包含若干“键——值”对。其中键是不重复的，每个键都被映射为一个值。如果键的类型是 K，值的类型是 V，我们记映射的类型为 $Map\ K\ V$ 或 `Map<K, V>`。非空映射包含 n 个关联（映射）关系：$k_1 \mapsto v_1, k_2 \mapsto v_2, ..., k_n \mapsto v_n$。当使用二叉搜索树实现映射时，我们限制 K 为有序集合。每个二叉树节点存储一对键、值。我们使用二叉搜索树的插入或更新算法将一对键、值关联起来。给定键 k，我们使用二叉搜索树的查找算法获取映射值。如果 k 不存在，则返回空。后面章节介绍的红黑树和 AVL 树也都可以用来实现映射数据结构。

2.9 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。

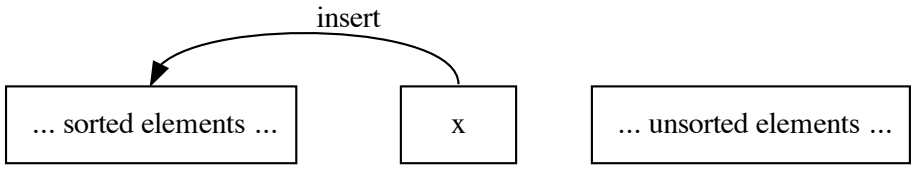


图 3.2: 不断将元素插入已序部分

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，因此插入排序的总体

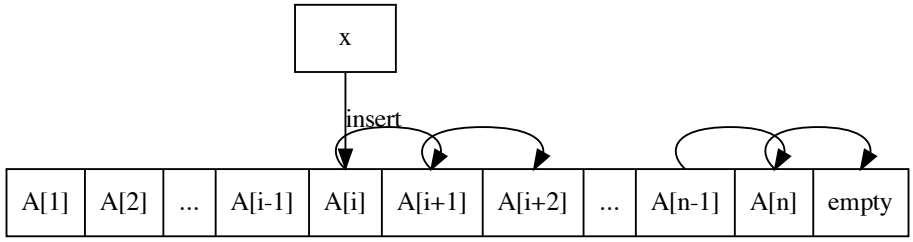


图 3.3: 将元素 x 插入数组 A 中的第 i 个位置

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
```

```
6:          return m                                   ▷ 重复元素
7:      else if A[m] < x then
8:          l ← m + 1
9:      else
10:         u ← m
11:   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，比较次数减少到了 $O(n \lg n)$，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
4:         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。能否使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。

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

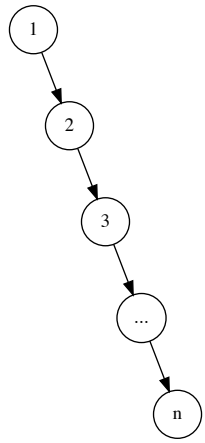


图 4.1: 不平衡的树

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:

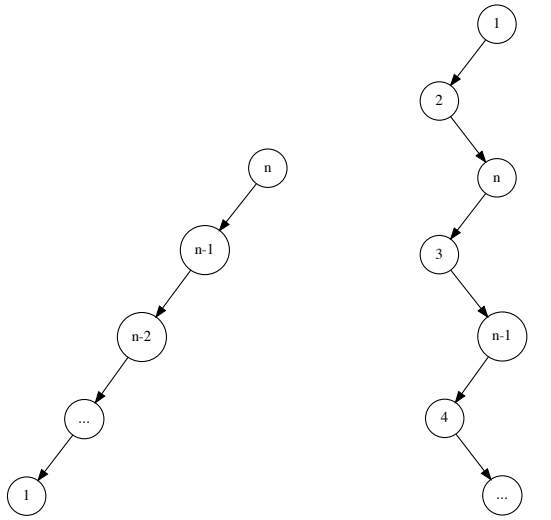
$$\begin{aligned} rotate_l\ (a, x, (b, y, c)) &= ((a, x, b), y, c) \\ rotate_l\ T &= T \end{aligned} \tag{4.1}$$

和

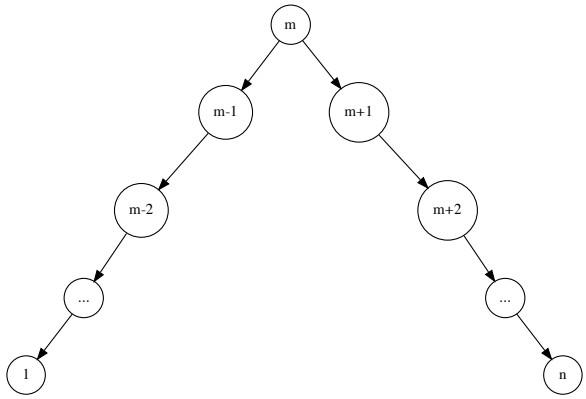
$$\begin{aligned} rotate_r\ ((a, x, b), y, c) &= (a, x, (b, y, c)) \\ rotate_r\ T &= T \end{aligned} \tag{4.2}$$

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 ← PARENT(x)
3:     y ← RIGHT(x)                                  ▷ 设 y ≠ NIL
4:     a ← LEFT(x)
5:     b ← LEFT(y)
6:     c ← RIGHT(y)
```



(a) (b)



(c)

图 4.2: 一些不平衡的二叉树

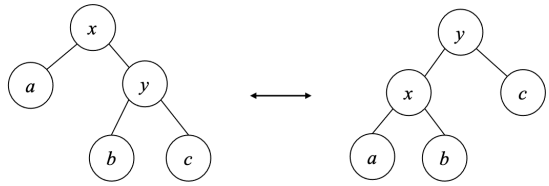


图 4.3: 树的左右旋转

```
    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。

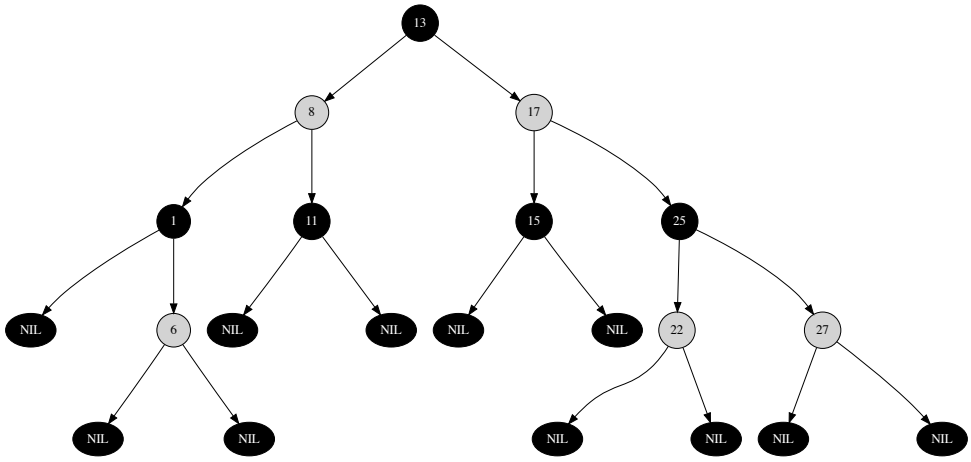


图 4.4: 红黑树

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4.5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。

下面的例子程序在二叉搜索树的基础上增加了颜色定义：

[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。

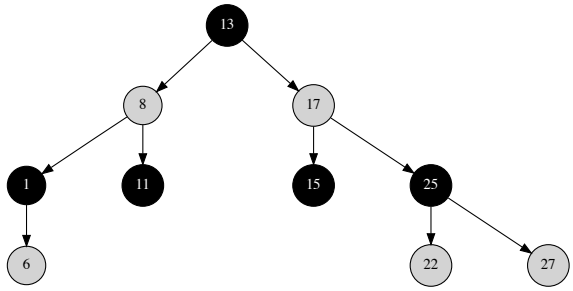


图 4.5: 隐藏 NIL 节点

```
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。

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

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:

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

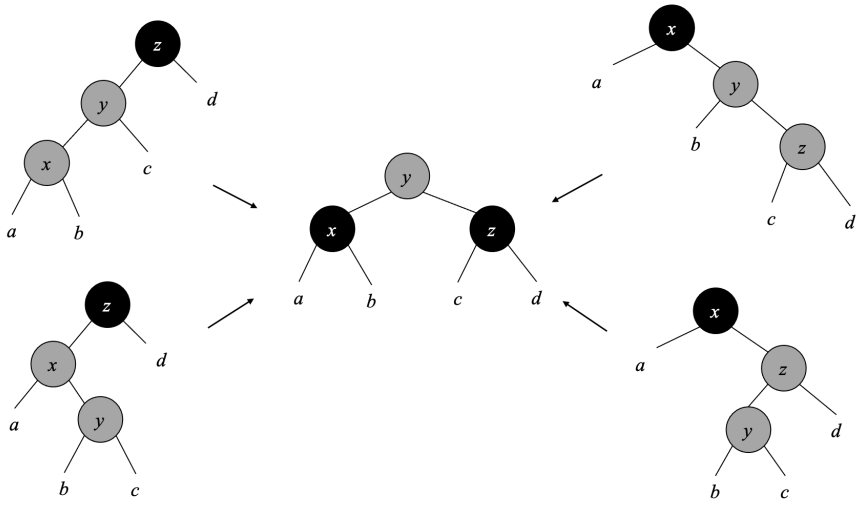


图 4.6: 插入后需要修复的四种情况

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 创建一个红色节点, 它的两个分枝都为空; 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, 15, 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。

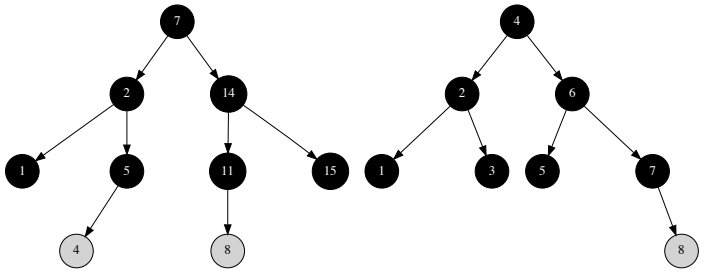


图 4.7: 插入产生的红黑树

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)
```

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 $\varnothing$）。删除时，第一步和普通二叉搜索树相同；如果被删除节点

[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

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:

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

del 接受一棵树和要删除的元素 k:

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

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。

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

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。

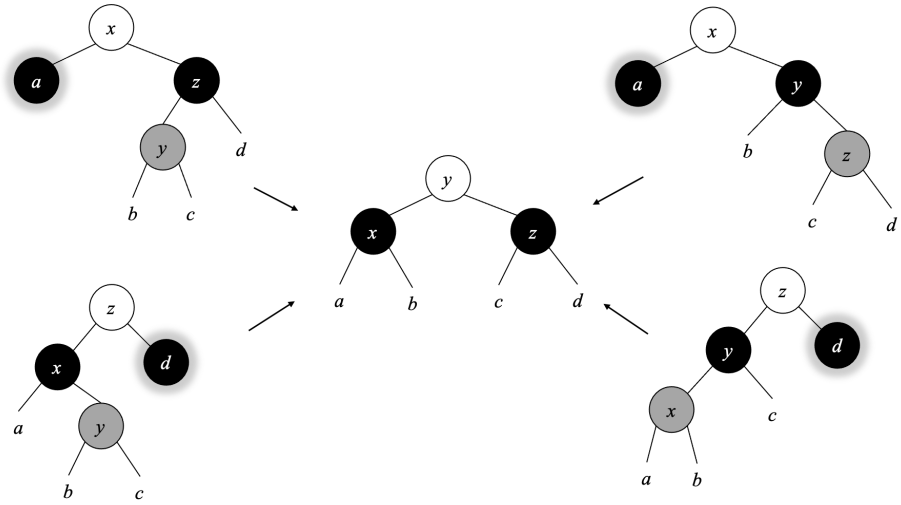


图 4.8: 4 种子情况可以修复为统一的形式

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。

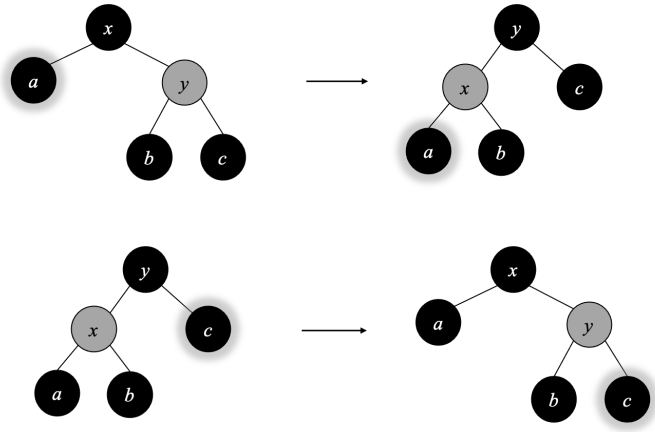


图 4.9: 双重黑色节点的兄弟节点为红色

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。

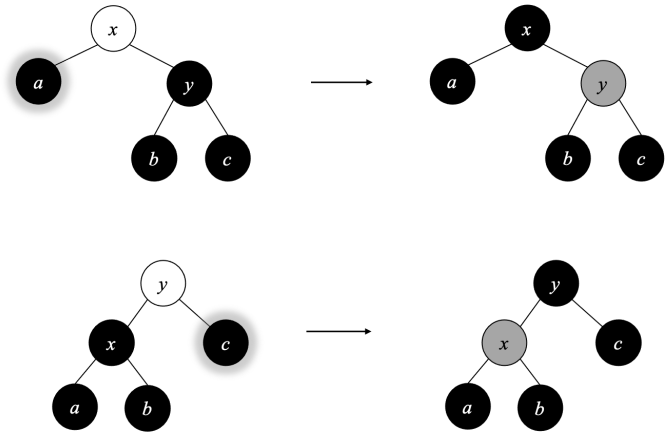


图 4.10: 将双重黑色向上传递

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:

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

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。

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> {
```

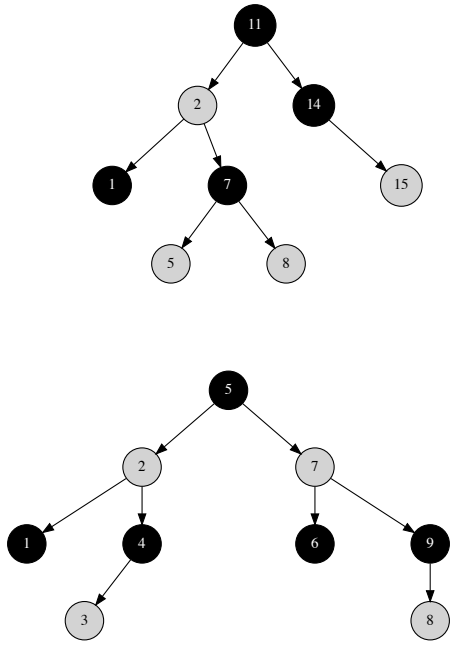


图 4.11: 命令式算法构建出的红黑树

```
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

为了解决平衡问题, 红黑树限制在某一路径上的节点数。AVL 树采用了更为直接方法:度量分枝间的差异。对于树 T, 定义:

$$\delta(T) = |r| - |l| \tag{5.1}$$

其中 $|T|$ 表示树 T 的高度, l、r 为左右子树。定义空树 $\delta(\varnothing) = 0$。如果每棵子树 T 都有 $\delta(T) = 0$, 则树是完全平衡的。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, 所有节点都不为空。$\delta(T)$ 的绝对值越小, 树越平衡。我们称 $\delta(T)$ 为二叉树的“平衡因子”。

5.1　定义

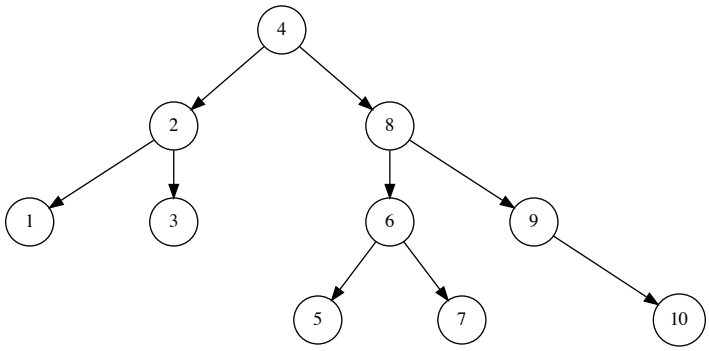


图 5.1: AVL 树

一棵二叉搜索树称为 AVL 树, 如果所有子树 T 都满足如下条件:

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

平衡因子 $\delta(T)$ 只能是 ± 1、0。图5.1给出了一棵 AVL 树的例子。如果树中有 n 个节点, 这一定义保证了树的高度 $h = O(\lg n)$。我们可以证明这个结论。一棵高为 h 的

AVL 树, 其节点数目并不是一个固定的值。当它是完全二叉树时, 含有的节点数目最多, 为 $2^h - 1$。我们关心它至少包含多少节点。定义 $N(h)$ 代表高度为 h 的 AVL 树的最小节点数目。我们有:

- 空树 $\varnothing$: $h = 0, N(0) = 0$;
- 只有一个节点的树: $h = 1, N(1) = 1$;

图5.2中给出了一个高度为 h 的 AVL 树 T。它包含三部分:元素 k 和左右子树 l、r。树的高度 h 和子树高度之间满足下面的关系:

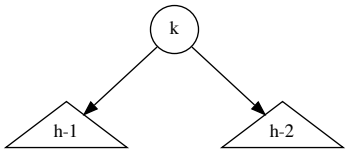


图 5.2: 高度为 h 的 AVL 树, 其中一棵子树高 $h-1$, 另外一棵的高度不小于 $h-2$

$$h = max(|l|, |r|) + 1 \tag{5.3}$$

因此必然存在一个子树的高度为 $h-1$。根据 AVL 树的定义, 有 $||l| - |r|| \leq 1$。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和加 1(节点 T 本身):

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

这一递归形式让我们联想起斐波那契数列。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的递归关系:

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

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 $\phi = \dfrac{\sqrt{5}+1}{2}$, 称为黄金分割比。

证明. 使用数学归纳法。当 $h = 0$ 或 1 时:

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

对于递推情况, 设 $N'(h) \geq \phi^h$。

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

□

由引理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。

插入和删除会改变树的结构, 导致平衡因子的绝对值超出 1, 需要通过修复使得 $|\delta| < 1$。传统的修复方法是树旋转。我们给出一种基于模式匹配的方法简化实现。思路类似于函数式的红黑树[13]。由于这种“改变——恢复”的策略, AVL 树也是一种自平衡二叉树。我们复用二叉搜索树的定义, 尽管平衡因子 δ 可以递归地求出, 为了方便, 我们在每个非空节点 $T = (l, k, r, \delta)$ 中保存平衡因子的值, 并在改变树结构时更新它[1]。下面的例子程序增加了一个整型变量 δ:

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

AVL 树的 $lookup$、max、min 等操作和二叉搜索树相同, 而插入和删除操作是特殊的。

5.2 插入

向 AVL 树中插入一个新元素时, 平衡因子的绝对值 $|\delta(T)|$ 可能超过 1。我们用类似红黑树修复的模式匹配方法恢复平衡。插入元素 x 后, 包含它的子树高度最多增加 1。我们需要沿着插入路径递归地更新平衡因子。定义插入结果为一对值 $(T', \Delta H)$, 其中 T' 为插入后的树, ΔH 为高度的增加值。我们将二叉搜索树的插入算法修改如下:

$$insert = fst \circ ins \tag{5.8}$$

其中 $fst\ (a, b) = a$ 返回一对值中的第一个。$ins(T, k)$ 将元素 x 插入到树 T 中:

$$\begin{aligned} ins\ \varnothing\ k &= ((\varnothing, k, \varnothing, 0), 1) \\ ins\ (l, k', r, \delta)\ k &= \begin{cases} k < k' : tree\ (ins\ l\ k)\ k'\ (r, 0)\ \delta \\ k > k' : tree\ (l, 0)\ k'\ (ins\ r, k)\ \delta \end{cases} \end{aligned} \tag{5.9}$$

[1] 也可以保存树的高度而非 δ[20]。

如果树为空 $\varnothing$, 结果为包含 k 的叶子节点, 平衡因子为 0, 高度增加 1。否则, 令 $T = (l, k', r, \delta)$。我们比较 k 和 k', 如果 $k < k'$, 我们递归地将 k 插入到左子树 l 中, 否则插入右子树 r。递归插入的结果也是一对值 $(l', \Delta l)$ 或 $(r', \Delta r)$。我们通过函数 $tree$ 调整平衡因子并更新高度, 它接受 4 个参数: $(l', \Delta l)$、k'、$(r', \Delta r)$、δ, 并产生结果 $(T', \Delta H)$。其中 T' 为新树, ΔH 定义如下:

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

它可以进一步分解为 4 种情况:

$$\begin{aligned}
\Delta H &= |T'| - |T| \\
&= 1 + max(|r'|, |l'|) - (1 + max(|r|, |l|)) \\
&= max(|r'|, |l'|) - max(|r|, |l|) \\
&= \begin{cases}
\delta \geq 0, \delta' \geq 0: & \Delta r \\
\delta \leq 0, \delta' \geq 0: & \delta + \Delta r \\
\delta \geq 0, \delta' \leq 0: & \Delta l - \delta \\
\text{否则}: & \Delta l
\end{cases}
\end{aligned} \tag{5.11}$$

其中 $\delta' = \delta(T') = |r'| - |l'|$, 是变化后的平衡因子。附录 B 给出了相关证明。在平衡调整前, 还需要确定新的平衡因子 δ'。

$$\begin{aligned}
\delta' &= |r'| - |l'| \\
&= |r| + \Delta r - (|l| + \Delta l) \\
&= |r| - |l| + \Delta r - \Delta l \\
&= \delta + \Delta r - \Delta l
\end{aligned} \tag{5.12}$$

使用树的高度变化和平衡因子, 就可进一步定义 (5.9) 中的函数 $tree$:

$$tree\ (l', \Delta l)\ k\ (r', \Delta r)\ \delta = balance\ (l', k, r', \delta')\ \Delta H \tag{5.13}$$

下面的例子程序实现了目前给出的结论:

```
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 = tree (l,  0) k (ins r) d

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

共有 4 种情况需要修复，如图5.3所示。平衡因子为 ± 2 超出了 $[-1, 1]$ 范围。我们将其统一调整为图中心的结构，使得 $\delta(y) = 0$。

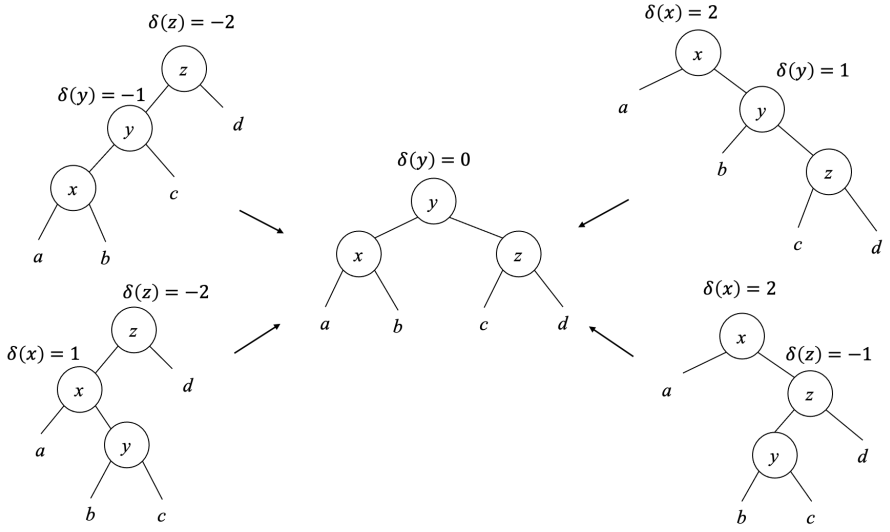


图 5.3: 将 4 种情况修复为统一形式

我们称这 4 种情况为左－左、右－右、右－左、左－右。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 $\delta'(x)$、$\delta'(y) = 0$、$\delta'(z)$。它们的关系如下。附录 B 给出了证明。

左－左：

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

右－右：

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

右－左和左－右：

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

利用模式匹配, 可将修复定义如下:

$$\begin{aligned}
balance\ (((a,x,b,\delta(x)),y,c,-1),z,d,-2)\ \Delta H &= ((a,x,b,\delta(x)),y,(c,z,d,0),0,\Delta H-1)\\
balance\ (a,x,(b,y,(c,z,d,\delta(z)),1),2)\ \Delta H &= ((a,x,b,0),y,(c,z,d,\delta(z)),0,\Delta H-1)\\
balance\ ((a,x,(b,y,c,\delta(y)),1),z,d,-2)\ \Delta H &= ((a,x,b,\delta'(x)),y,(c,z,d,\delta'(z)),0,\Delta H-1)\\
balance\ (a,x,((b,y,c,\delta(y)),z,d,-1),2)\ \Delta H &= ((a,x,b,\delta'(x)),y,(c,z,d,\delta'(z)),0,\Delta H-1)\\
balance\ T\ \Delta H &= (T,\Delta H)
\end{aligned} \tag{5.17}$$

其中 $\delta'(x)$ 和 $\delta'(z)$ 按照式 (B.17) 定义。如果没有匹配任何模式, 最后一行保持树不变。下面是相应的例子程序:

```
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
balance t d = (t, d)
```

插入算法的复杂度和树的高度成正比, 根据式 (5.7), 对于 n 个节点的树, $insert$ 的复杂度为 $O(\lg n)$。

验证

为了验证 AVL 树, 我们需要检查两点: 是否是二叉搜索树; 对于每棵子树 T, 式 (5.2): $\delta(T) \leq 1$ 是否成立。下面的函数递归地检查子树间的高度差:

$$\begin{aligned}
avl?\ \varnothing &= True\\
avl?\ T &= avl?\ l\ \wedge avl?\ r\ \wedge ||r|-|l|| \leq 1
\end{aligned} \tag{5.18}$$

其中 l、r 分别是左右子树, 高度递归计算如下:

$$\begin{aligned}
|\varnothing| &= 0\\
|T| &= 1 + max(|r|,|l|)
\end{aligned} \tag{5.19}$$

下面的例子程序实现了 AVL 树高度的检查:

```
isAVL Empty = True
isAVL (Br l _ r _) = 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

1. 我们只验证了 AVL 树的高度性质, 完成验证程序检查一棵二叉树是否是 AVL 树。

5.3 AVL 树的命令式算法 ★

为了完整, 本节给出 AVL 树的命令式算法。和红黑树的命令式算法相似, 我们先按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。插入到右侧可能使 δ 增加 1, 插入左侧可能使 δ 减少 1。我们从 x 开始, 自底向上修复平衡, 直到根节点。记新的平衡因子为 δ', 共有 3 种情况:

- $|\delta| = 1$、$|\delta'| = 0$。插入后树处于平衡状态。父节点的高度没有变化。
- $|\delta| = 0$、$|\delta'| = 1$。左右子树之一的高度增加了, 需要继续向上检查平衡。
- $|\delta| = 1$、$|\delta'| = 2$。需要旋转以修复平衡。

```
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
13:         return T
14:     else if |δ| = 0 and |δ'| = 1 then               ▷ 继续自底向上更新
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
19:                 δ(P) ← 0                            ▷ 根据式 (B.6)
20:                 δ(R) ← 0
21:                 T ← LEFT-ROTATE(T, P)
22:             if δ(R) = −1 then                       ▷ 右-左
23:                 δ_y ← δ(LEFT(R))                    ▷ 根据式 (B.17)
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
37:                 δ(P) ← 0
38:                 δ(L) ← 0
39:                 RIGHT-ROTATE(T, P)
```

```
            else                                                ▷ 左-右
                δ_y ← δ(Right(L))
                if δ_y = 1 then
                    δ(L) ← −1
                else
                    δ(L) ← 0
                δ(Right(L)) ← 0
                if δ_y = −1 then
                    δ(P) ← 1
                else
                    δ(P) ← 0
                Left-Rotate(T, L)
                Right-Rotate(T, P)
            break
    return T
```

除了旋转, 还需要更新平衡因子 δ。右－右和左－左情况需要进行一次旋转; 而右－左和左－右需要进行两次旋转。我们略过了 AVL 树的删除算法, 附录 B 给出了删除的实现。

5.4 小结

AVL 树是 1962 年由 Adelson-Velskii 和 Landis[18]、[19] 提出的, 并以两位作者的名字命名。它比红黑树更早。AVL 树和红黑树都是自平衡二叉搜索树, 大多数操作的复杂度都是 $O(\lg n)$。式 (5.7) 使得 AVL 树的平衡性更为严格。在大量查询的情况下, 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。很多程序库使用红黑树作为自平衡二叉搜索树的内部实现, AVL 树同样也可以直观、高效地解决平衡问题。

5.5 附录:例子程序

AVL 树的定义:

```
data Node<T> {
    int delta
    T key
    Node<T> left
    Node<T> right
    Node<T> parent
}
```

平衡修复:

```
Node<T> insertFix(Node<T> t, Node<T> x) {
    while (x.parent ≠ null ) {
        var (p, l, r) = (x.parent, x.parent.left, x.parent.right)
        var d1 = p.delta
        var d2 = if x == parent.left then d1 - 1 else d1 + 1
        p.delta = d2

        if abs(d1) == 1 and abs(d2) == 0 {
            return t
        } else if abs(d1) == 0 and abs(d2) == 1 {
            x = p
        } else if abs(d1) == 1 and abs(d2) == 2 {
            if d2 == 2 {
                if r.delta == 1 {    //Right-right
                    p.delta = 0
                    r.delta = 0
                    t = rotateLeft(t, p)
                } else if r.delta == -1 {    //Right-Left
                    var dy = r.left.delta
                    p.delta = if dy == 1 then -1 else 0
                    r.left.delta = 0
                    r.delta = if dy == -1 then 1 else 0
                    t = rotateRight(t, r)
                    t = rotateLeft(t, p)
                }
            } else if d2 == -2 {
                if l.delta == -1 {    //Left-left
                    p.delta = 0
                    l.delta = 0
                    t = rotateRight(t, p)
                } else if l.delta == 1 {    //Left-right
                    var dy = l.right.delta
                    l.delta = if dy == 1 then -1 else 0
                    l.right.delta = 0
                    p.delta = if dy == -1 then 1 else 0
                    t = rotateLeft(t, l)
                    t = rotateRight(t, p)
                }
            }
            break
        }
    }
    return t
}
```

第六章　基数树

排序二叉树将信息存储在节点中。我们可以用边 (edge) 来携带信息么? 基数树 (Radix tree), 包括 trie、前缀树、后缀树就是根据这一思路设计出的数据结构。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。

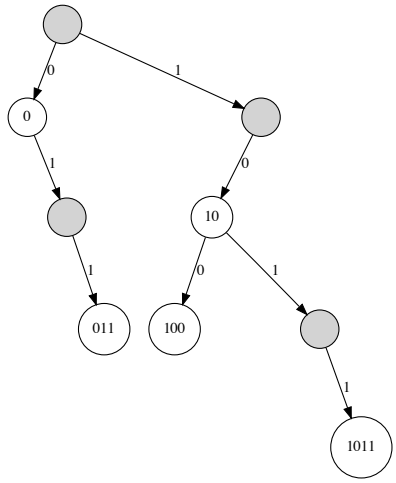


图 6.1: 基数树

图6.1展示了一棵基数树。它包含了二进制串 1011、10、011、100、0。如果查找二进制数 $k = (b_0 b_1 ... b_n)_2$, 我们首先检查左侧的最高位 b_0。若为 0, 则转向左子树继续查找; 若为 1, 则转向右子树。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。我们并不需要在节点中存储键(key), 这一信息由边来代表。图6.1标注在节点中的键仅仅是为了示意。对于整数类型的键, 我们可以使用二进制, 并利用位运算进行操作。

6.1　整数 trie

我们称图6.1所示的数据结构为 *binary trie*。Trie 是 Edward Fredkin 在 1960 年提出的。它来自英文单词 re**trie**val。Fredkin 将其读作/'tri:/, 但其他人读作/'trai/(和英文单词 try 的发音相同)[24]。有些情况下 trie 也被称为前缀树, 在本章中, trie 和前

缀树分指不同的数据结构。一棵 binary trie 是一种特殊的二叉树，每个键的位置由它的二进制位来决定。0 表示“向左”，1 表示“向右”[21]。考虑图6.2中的 trie，3 个不同串“11”、“011”、“0011” 代表同一个十进制整数 3。

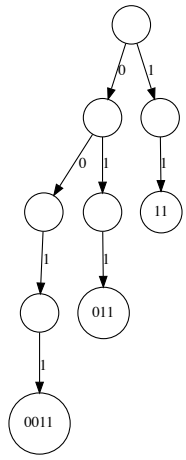


图 6.2: 大端(big-endian)trie

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 32 层的树。为了解决这一问题，Okasaki 建议使用小端整数[21]。二进制数的最高位(MSB)通常在左边，最低位(LSB)在右边。这种形式称为大端整数，反之最高位在右边称为小端整数。使用小端，1 表示为 $(1)_2$、2 表示为 $(01)_2$、3 表示为 $(11)_2$……

6.1.1 定义

我们可以重用二叉树的定义。一个节点要么为空，要么包含左右子树和一个值(值可以为空)左子树编码为 0，右子树编码为 1。

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

对于 binary trie 中的任一节点，其对应的整数键是由节点的位置唯一确定的。因此我们无需在节点中存储键，而只需存储值。键的类型被固定为整数，如果值的类型为 A，则树的类型为 $IntTrie\ A$。

6.1.2 插入

当插入整数键 k 和值 v 时，我们将 k 转换成二进制。如果 k 是偶数，最低位是 0，我们递归向左子树插入；如果 k 是奇数，最低位是 1，我们递归向右子树插入。接下来我们将 k 除以 2 取整以去掉最低位。对于非空的 trie 树 $T = (l, v', r)$，其中 l、r 是左

右子树, v' 是值(可为空), 函数 $insert$ 的定义如下:

$$\begin{aligned} insert\ \varnothing\ k\ v &= insert(\varnothing, Nothing, \varnothing)\ k\ v \\ insert\ (l, v', r)\ 0\ v &= (l, Just\ v, r) \\ insert\ (l, v', r)\ k\ v &= \begin{cases} even(k): & (insert\ l\ \frac{k}{2}\ v, v', r) \\ odd(k): & (l, v', insert\ r\ \lfloor\frac{k}{2}\rfloor\ v) \end{cases} \end{aligned} \tag{6.1}$$

如果 $k = 0$, 我们将 v 存入节点。如果 $T = \varnothing$, 结果为 $(\varnothing, Just\ v, \varnothing)$。只要 $k \neq 0$, 我们就根据 k 的奇偶性前进, 遇到 $\varnothing$ 就建立一个空叶子节点 $(\varnothing, Nothing, \varnothing)$。如果 k 已经存在, 这一算法覆盖以前的值。我们也可以用一个列表存储多个值, 并将 v 添加到列表中。图6.1的例子是依次插入映射 $\{1 \to a, 4 \to b, 5 \to c, 9 \to d\}$ 的结果。下面的例子程序实现了 $insert$ 函数:

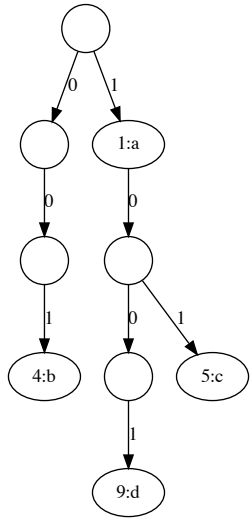


图 6.3: 小端整数 trie, 包含映射: $\{1 \to a, 4 \to b, 5 \to c, 9 \to d\}$

```
insert Empty k x = insert (Branch Empty Nothing Empty) k x
insert (Branch l v r) 0 x = Branch l (Just x) r
insert (Branch l v r) k x | even k    = Branch (insert l (k `div` 2) x) v r
                          | otherwise = Branch l v (insert r (k `div` 2) x)
```

为了判定一个数的奇偶性, 我们可以判断它除以 2 的余数（模 2）是否为 0: $even(k) = (k \bmod 2 = 0)$, 或者使用位运算, 如: `(k & 0x1) == 0`。我们也可以消除递归用循环实现插入算法:

```
1: function INSERT(T, k, v)
2:     if T = NIL then
3:         T ← EMPTY-NODE                          ▷ (NIL, Nothing, NIL)
4:     p ← T
5:     while k ≠ 0 do
```

```
        if Even?(k) then
            if Left(p) = NIL then
                Left(p) ← Empty-Node
            p ← Left(p)
        else
            if Right(p) = NIL then
                Right(p) ← Empty-Node
            p ← Right(p)
        k ← ⌊k/2⌋
    Value(p) ← v
    return T
```

Insert 接受 3 个参数:trie 树 T、要插入的键 k 和相应的数据 v。对于有 m 位的二进制整数 k, 这一算法访问 trie 中的 m 层, 时间复杂度为 $O(m)$。

6.1.3 查找

在一棵非空的小端整数 trie 中查找 k 时, 若 $k = 0$, 则返回根节点中存储的数据。否则根据最后一位是 0 还是 1, 对左右子树进行递归查找。

$$\begin{aligned} lookup\ \varnothing\ k &= Nothing \\ lookup\ (l, v, r)\ 0 &= v \\ lookup\ (l, v, r)\ k &= \begin{cases} even(k): & lookup\ l\ \frac{k}{2} \\ odd(k): & lookup\ r\ \lfloor\frac{k}{2}\rfloor \end{cases} \end{aligned} \tag{6.2}$$

下面的例子程序实现了 *lookup* 函数:

```
lookup Empty _ = Nothing
lookup (Branch _ v _) 0 = v
lookup (Branch l _ r) k | even k    = lookup l (k `div` 2)
                        | otherwise = lookup r (k `div` 2)
```

我们也可以消除递归实现迭代式的查找算法:

```
function Lookup(T, k)
    while k ≠ 0 and T ≠NIL do
        if Even?(k) then
            T ← Left(T)
        else
            T ← Right(T)
        k ← ⌊k/2⌋
    if T ≠ NIL then
```

```
9:          return VALUE(T)
10:     else
11:         return NIL
```

对于有 m 位的整数 k, *lookup* 函数的复杂度为 $O(m)$。

练习 6.1

1. 是否可以将定义 `Branch (IntTrie a) (Maybe a) (IntTrie a)` 变为 `Branch (IntTrie a) a (IntTrie a)`, 如果值不存在返回 *Nothing*, 否则返回 *Just v*?

6.2 整数前缀树

Trie 的缺点是空间消耗大。在图6.1的例子中, 只有 4 个节点存有数据, 其它 5 个节点都是空的。空间利用率不足 50%。为了提高空间利用率, 我们可以将链接在一起的节点压缩成一个。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]。当键是整数时, 我们称之为整数前缀树, 或者在不引起歧义的情况下简称为整数树。Okasaki 给出了整数前缀树的实现[21]。将图6.3中链接在一起的节点合并后, 可以得到一棵如图6.4所示的树。

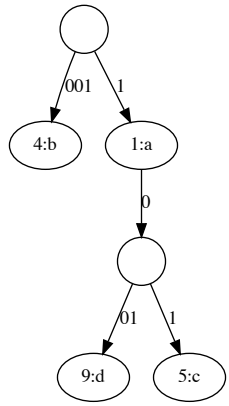


图 6.4: 小端整数树实现的映射 $\{1 \to a, 4 \to b, 5 \to c, 9 \to d\}$

在整数树中, 分枝节点对应的键是它所有子树的公共前缀。这些子树的键在其公共前缀的下一位开始不同。这样整数前缀树消除了不必要的存储空间。

6.2.1 定义

整数前缀树是一种特殊的二叉树。它或者为空, 或者是一个如下的节点:

- 叶子节点: 包含一个整数键 k 和一个值 v;

- 分枝节点:其左右子树共有一个二进制**最长公共前缀**,左子树的下一位是 0,而右子树的下一位是 1。

下面的例子代码定义了整数前缀树。分枝节点包含 4 个部分:最长公共前缀、一个掩码表明从哪一位开始分枝出子树、左右子树。掩码为 $m = 2^n$ 的形式,其中整数 $n \geq 0$。所有低于 n 位的二进制位都不属于公共前缀。

```
data IntTree a = Empty
               | Leaf Int a
               | Branch Int Int (IntTree a) (IntTree a)
```

6.2.2 插入

当向树 T 插入整数 y 时,若 T 为空,我们用 y 创建一个叶子节点;如果 T 本身只包含一个叶子节点 x,我们创建一个分枝,并将两个叶子节点 x 和 y 分别设为左右子树。为了确定左右,我们需要找到 x 和 y 的最长公共前缀 p。例如,$x = 12 = (1100)_2$、$y = 15 = (1111)_2$,则 $p = (11oo)_2$,其中 o 表示我们不关心的二进制位,我们使用一个掩码整数 m 来去掉(mask)这些位。在本例中,$m = 4 = (100)_2$,最长公共前缀 p 后面的一位代表 2^1。x 中这一位是 0,而 y 中这一位是 1。因此 x 是左子树,而 y 是右子树。如图6.5所示。

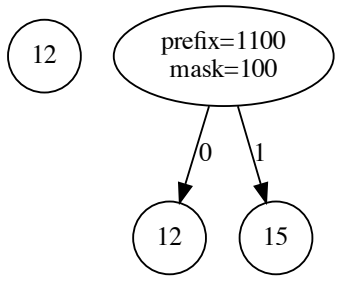
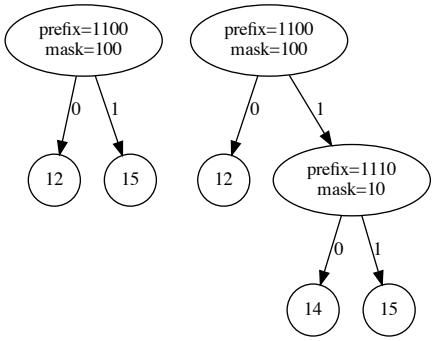


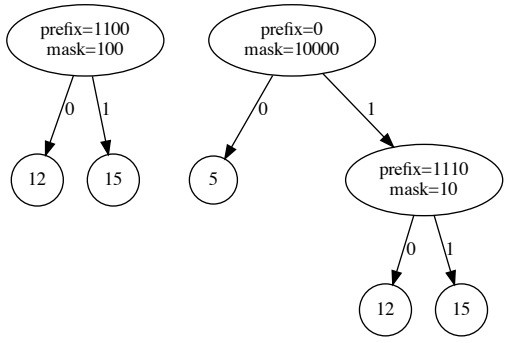
图 6.5: 左:T 是叶子节点 12;右:插入 15 后

如果树 T 既不为空,也不是叶子节点,我们需要先比较 y 是否和 T 的最长公共前缀 p 匹配。如果匹配,则根据下一位是 0 或 1 递归地在左、右子树中插入。例如,若将 $y = 14 = (1110)_2$ 插入图6.5所示的树,由于最长公共前缀 $p = (11oo)_2$,并且接下来的一位(2^1 位)是 1,我们递归地将 y 插入右子树。如果 y 和最长公共前缀 p 不匹配,我们需要分出一个新叶子节点。如图6.6所示。

如果整数键为 k 值为 v,令叶子节点为 (k, v)。如果是分枝节点,则记为 (p, m, l, r),其中 p 是最长公共前缀,m 是掩码,l 和 r 分别是左右子树。下面的 $insert$ 函数包含



(a) 插入 $14 = (1110)_2$, 它和最长公共前缀 $p = (1100)_2$ 匹配, 递归插入右子树。



(b) 插入 $5 = (101)_2$, 它和最长公共前缀 $p = (1100)_2$ 不匹配, 分出一个新叶子节点。

图 6.6: 根节点为分枝

了上述三种情况：

$$
\begin{aligned}
insert\ \varnothing\ k\ v &= (k, v) \\
insert\ (k, v')\ k\ v &= (k, v) \\
insert\ (k', v')\ k\ v &= join\ k\ (k, v)\ k'\ (k', v') \\
insert\ (p, m, l, r)\ k\ v &= \begin{cases} match(k, p, m): & \begin{cases} zero(k, m): & (p, m, insert\ l\ k\ v) \\ otherwise: & (p, m, insert\ r\ k\ v) \end{cases} \\ otherwise: & join\ k\ (k, v)\ p\ (p, m, l, r) \end{cases}
\end{aligned} \tag{6.3}
$$

当 $T = \varnothing$ 为空时，我们创建一个叶子节点；如果键相同，我们用新值覆盖此前的值。函数 $match(k, p, m)$ 检查整数 k 和最长公共前缀 p 在掩码 m 下是否相同：$mask(k, m) = p$，其中 $mask(k, m) = \overline{m-1} \& k$。它先对 $m-1$ 按位取反，然后和 k 按位与。函数 $zero(k, m)$ 检查掩码 m 之后的二进制位是 0 还是 1。我们将 m 向右移动 1 位，然后和 k 按位与：

$$
zero(k, m) = k \& (m >> 1) \tag{6.4}
$$

函数 $join(p_1, T_1, p_2, T_2)$ 接受两个前缀和两棵树。它从 p_1、p_2 抽出最长公共前缀 $(p, m) = LCP(p_1, p_2)$，创建一个新分枝，并将 T_1、T_2 设为子树：

$$
join(p_1, T_1, p_2, T_2) = \begin{cases} zero(p1, m): & (p, m, T_1, T_2) \\ otherwise: & (p, m, T_2, T_1) \end{cases} \tag{6.5}
$$

为了计算最长公共前缀，我们先对 p_1、p_2 按位计算异或，然后数出最高位 $highest(xor(p_1, p_2))$：

$$
\begin{aligned}
highest(0) &= 0 \\
highest(n) &= 1 + highest(n >> 1)
\end{aligned}
$$

接下来我们产生掩码 $m = 2^{highest(xor(p_1, p_2))}$。最长公共前缀 p 可以用掩码 m 和 p_1、p_2 中的任何一个得出。例如 $p = mask(p_1, m)$。下面的例子程序实现了 $insert$ 函数：

```
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
      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

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 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)

zero x m = x .&. (shiftR m 1) == 0

match k p m = (mask k m) == p
```

我们也可以用命令式方法实现 *insert*:

```
1: function INSERT(T, k, v)
2:     if T = NIL then
3:         return CREATE-LEAF(k, v)
4:     y ← T
5:     p ← NIL
6:     while y is not leaf, and MATCH(k, PREFIX(y), MASK(y)) do
7:         p ← y
8:         if ZERO?(k, MASK(y)) then
9:             y ← LEFT(y)
10:        else
11:            y ← RIGHT(y)
12:    if y is leaf, and k = KEY(y) then
13:        VALUE(y) ← v
14:    else
15:        z ← BRANCH(y, CREATE-LEAF(k, v))
16:        if p = NIL then
17:            T ← z
18:        else
19:            if LEFT(p) = y then
20:                LEFT(p) ← z
21:            else
22:                RIGHT(p) ← z
23:    return T
```

其中 $\text{BRANCH}(T_1, T_2)$ 创建一个新分枝, 抽出最长公共前缀, 并将 T_1 和 T_2 设为子树。

```
1: function BRANCH(T_1, T_2)
2:     T ← EMPTY-NODE
3:     (PREFIX(T), MASK(T)) ← LCP(PREFIX(T_1), PREFIX(T_2))
4:     if ZERO?(PREFIX(T_1), MASK(T)) then
5:         LEFT(T) ← T_1
6:         RIGHT(T) ← T_2
7:     else
8:         LEFT(T) ← T_2
9:         RIGHT(T) ← T_1
10:    return T

11: function ZERO?(x, m)
12:    return (x&⌊m/2⌋) = 0
```

函数 LCP 获取两个整数的最长公共前缀:

```
1: function LCP(a, b)
2:     d ← xor(a, b)
3:     m ← 1
4:     while d ≠ 0 do
5:         d ← ⌊d/2⌋
6:         m ← 2m
7:     return (MASKBIT(a, m), m)

8: function MASKBIT(x, m)
9:     return x&¬(m − 1)
```

图6.7展示了使用插入算法构造的前缀树。虽然整数前缀树压缩了链接在一起的节点, 但获取最长前缀仍然需要线性扫描二进制位, 对 m 位整数, 插入算法的复杂度是 $O(m)$。

6.2.3 查找

当查找整数 k 时,如果树 $T = \varnothing$ 为空,或者是一个叶子节点 $T = (k', v)$ 但 $k \neq k'$, 则 k 不存在;如果 $k = k'$, 则 v 为查找结果。如果 $T = (p, m, l, r)$ 是一个分枝节点, 我们需要检查公共前缀 p 和 k 在掩码 m 下是否匹配, 并根据下一位是 0/1 递归地在子

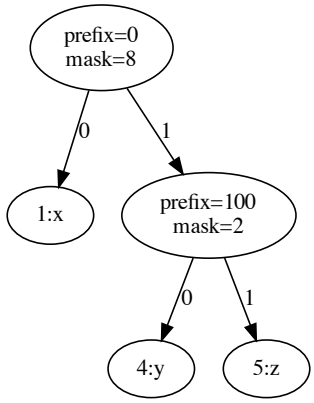


图 6.7: 插入映射 $1 \to x, 4 \to y, 5 \to z$ 到大端整数前缀树

树 l 或 r 中查找。如果不匹配公共前缀 p, 则 k 不存在。

$$
\begin{array}{rcl}
lookup\ \varnothing\ k & = & Nothing \\
lookup\ (k', v)\ k & = & \begin{cases} k = k' : & Just\ v \\ otherwise : & Nothing \end{cases} \\
lookup\ (p, m, l, r)\ k & = & \begin{cases} match(k, p, m) : & \begin{cases} zero(k, m) : & lookup\ l\ k \\ otherwise : & lookup\ r\ k \end{cases} \\ otherwise : & Nothing \end{cases}
\end{array}
\tag{6.6}
$$

我们也可以消除递归改用迭代的方式实现查找。

```
1:  function LOOK-UP(T, k)
2:      if T = NIL then
3:          return NIL
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 VALUE(T)
11:     else
12:         return NIL
```

对于有 m 位的二进制整数, *lookup* 算法的复杂为 $O(m)$。

练习 6.2

1. 编写程序实现整数前缀树的 *lookup* 算法。

2. 实现整数 trie 和整数树的前序遍历，仅输出值不为空的节点键。结果有何规律？

6.3 Trie

在整数 trie 和整数前缀树的基础上，我们可以把键的类型从整数扩展到列表。其中一个特例是作为字符列表的字符串。前缀树和 trie 可以作为文本处理的有力工具。

6.3.1 定义

当键从二进制 0/1 扩展到通用列表时，树结构也自然从二叉树扩展为多分枝树。拿英语来说，一共有 26 个字符，如果忽略大小写，分枝的个数可以达到 26，如图6.8所示。

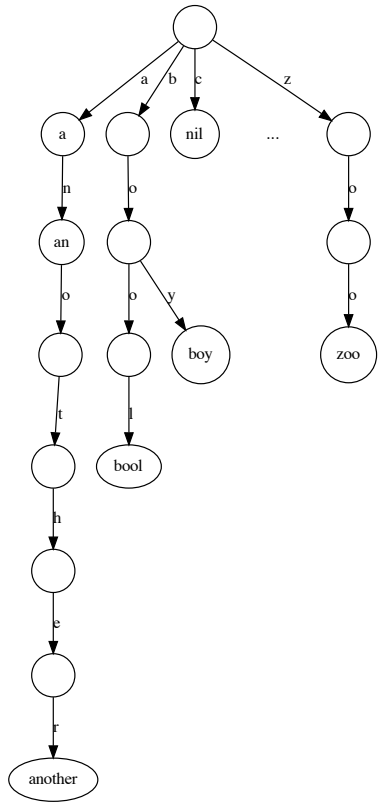


图 6.8: 含有多达 26 个分枝的 trie，包含键：a、an、another、bool、boy、zoo

并非所有的 26 棵子树都包含数据。在图6.8的根节点分枝中，只有代表 a、b、z 的 3 棵子树不为空。其它分枝，例如代表 c 的子树，全部是空的。我们可以将这些空子树隐藏起来。如果区分大小写或者需要进一步将类型从字符串扩展到抽象列表，我们可以使用 map 等数据结构来处理动态数目的分枝。

一棵 trie 或者为空，或者是一个节点，包含以下两种情况：

1. 值为 v 的叶子节点, 不含有子树;
2. 分枝节点, 包含值 v 和多棵子树。每棵子树对应类型 K 中的某个值 k。

若值的类型为 V, 我们记 trie 的类型为 $Trie\ K\ V$。下面的例子程序定义了 trie:

```
data Trie k v = Trie { value :: Maybe v
                     , subTrees :: [(k, Trie k v)]}
```

内容为空的树被定义为: $(Nothing, \varnothing)$

6.3.2 插入

考虑插入一对键、值到 trie 中, 其中键为若干元素的列表。令 trie 为 $T = (v, ts)$, 其中 v 是树中存储的值, $ts = \{c_1 \mapsto T_1, c_2 \mapsto T_2, ..., c_m \mapsto T_m\}$ 包含字符和子树间的映射。元素 c_i 映射到子树 T_i。我们可以使用关联列表 $[(c_1, T_1), (c_2, T_2), ..., (c_m, T_m)]$ 或者自平衡树实现映射(见第 4、5 章)。

$$\begin{aligned} insert\ (v, ts)\ \varnothing\ v' &= (v', ts) \\ insert\ (v, ts)\ (k : ks)\ v' &= (v, ins\ ts) \end{aligned} \tag{6.7}$$

如果键为空, 我们覆盖掉以前的值; 否则, 我们取出第一个元素 k, 找到映射到 k 的子树, 并递归将 ks 和 v' 插入:

$$\begin{aligned} ins\ \varnothing &= [k \mapsto insert\ (Nothing, \varnothing)\ ks\ v'] \\ ins\ ((c \mapsto t) : ts) &= \begin{cases} c = k : & (k \mapsto insert\ t\ ks\ v') : ts \\ otherwise : & (c \mapsto t) : (ins\ ts) \end{cases} \end{aligned} \tag{6.8}$$

如果没有子树映射到 k, 我们就新建一棵空子树 $t = (Nothing, \varnothing)$, 并将 k 映射到其上; 否则, 我们找到映射到 k 的子树 t, 递归地向 t 中插入 ks 和 v'。下面的例子程序实现了 $insert$ 函数, 它使用关联列表保存映射。

```
insert (Trie _ ts) [] x = Trie (Just x) ts
insert (Trie v ts) (k:ks) x = Trie v (ins ts) where
    ins [] = [(k, insert empty ks x)]
    ins ((c, t) : ts) = if c == k then (k, insert t ks x) : ts
                        else (c, t) : (ins ts)

empty = Trie Nothing []
```

我们也可以消除递归, 实现迭代方式的 $insert$ 函数:

```
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 SUB-TREES(p)[c] = NIL then
7:              SUB-TREES(p)[c] ← EMPTY-NODE
8:          p ← SUB-TREES(p)[c]
9:      VALUE(p) ← v
10:     return T
```

若键的类型为 $[K]$(K 的列表), K 是包含 m 个元素的有限集, 键的长度为 n, 则插入算法的复杂度为 $O(mn)$。当键是小写英文字符串时, $m = 26$, 插入操作的复杂度和字符串长度成正比。

6.3.3 查找

在 $T = (v, ts)$ 中查找一个非空键 $(k : ks)$ 时, 我们从第一个元素 k 开始, 如果存在映射到 k 的子树 T', 则接下来递归地在 T' 中查找 ks。当键为空时, 返回当前节点的值作为结果:

$$\begin{aligned} lookup\ \varnothing\ (v, ts) &= v \\ lookup\ (k : ks)\ (v, ts) &= \begin{cases} lookup_l\ k\ ts = Nothing : & Nothing \\ lookup_l\ k\ ts = Just\ t : & lookup\ ks\ t \end{cases} \end{aligned} \tag{6.9}$$

其中函数 $lookup_l$ 的定义见第一章。它在关联列表中查找键是否存在。下面是相应的迭代实现:

```
1: function LOOK-UP(T, key)
2:     if T = NIL then
3:         return Nothing
4:     for each c in key do
5:         if SUB-TREES(T)[c] = NIL then
6:             return Nothing
7:         T ← SUB-TREES(T)[c]
8:     return VALUE(T)
```

查找算法的复杂度为 $O(mn)$, 其中 n 是键的长度, m 是元素所在集合的大小。

练习 6.3

1. 使用自平衡二叉树(如红黑树或 AVL 树)实现映射 map 数据结构, 用以存储子树的映射。我们称其为 $MapTrie$ 或 $MapTree$。它们的插入和查找算法性能是怎样的?

6.4 前缀树

Trie 的空间利用率很低。我们可以用同样的方法压缩链接在一起的节点, 这样就得到了前缀树。

6.4.1 定义

一个前缀树节点 t 包含两部分:一个可为空的值 v、零个或若干子前缀树。每棵子树 t_i 对应到一个列表 s_i。列表和子树的映射关系记为 $[s_i \mapsto t_i]$。这些列表拥有共同的前缀 s, 而 s 映射到节点 t。也就是说 s 是 $s + s_1$、$s + s_2$……的最长公共前缀。对于任何 $i \neq j$, 列表 s_i、s_j 不存在非空的公共前缀。将图6.8中链接在一起的节点压缩起来, 可以得到如图6.9的前缀树。

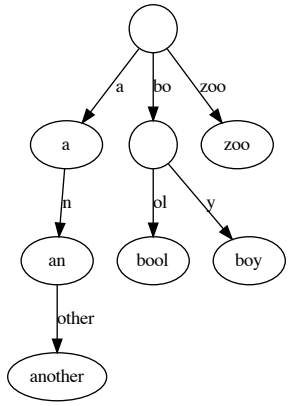


图 6.9: 一棵前缀树, 包含键:a、an、another、bool、boy、zoo

下面的例子程序定义了前缀树:

```
data PrefixTree k v = PrefixTree { value :: Maybe v
                                 , subTrees :: [([k], PrefixTree k v)]}
```

我们记前缀树为 $t = (v, ts)$。特别地,$(Nothing, \varnothing)$ 表示内容为空的节点;$(Just\, v, \varnothing)$ 表示值为 v 的叶子节点。

6.4.2 插入

插入字符串 s 时, 若前缀树为空, 我们为 s 创建一个叶子节点, 如图6.10(a) 所示。如果 s 和某个子树 t_i 相对应的 s_i 存在公共前缀, 我们创建一个新叶子节点 t_j, 抽出公共前缀, 并将其映射到一个新分枝 t' 上, 然后令 t_i、t_j 分别为 t' 的两棵子树。如图6.10(b) 所示。这里有两种特殊情况:s 为 s_i 的前缀, 如图6.10(c) → (e);以及 s_i 为 s 的前缀, 如图6.10(d) → (e)。

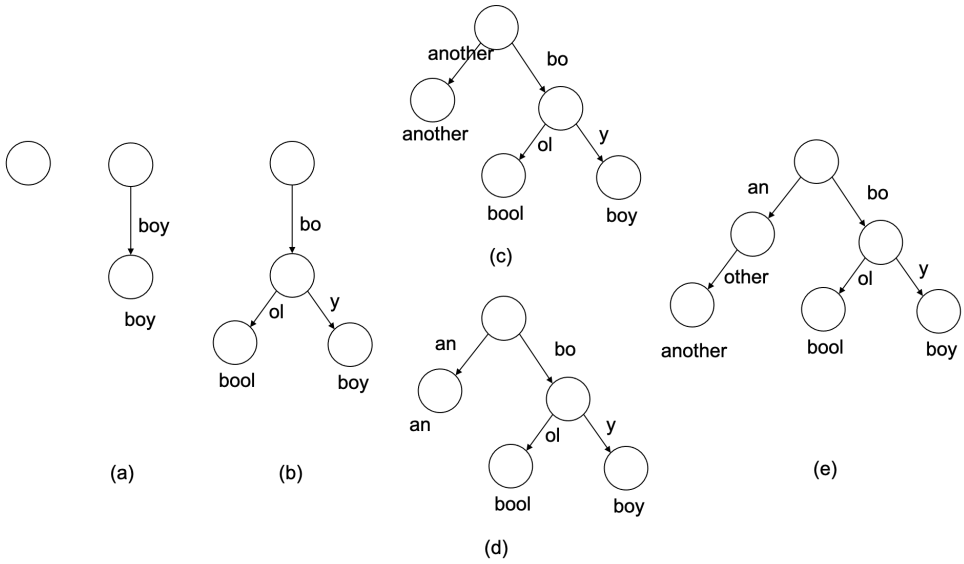


图 6.10: (a) 向空树插入 boy;(b) 插入 bool, 创建新分枝;(c) 向 (b) 插入 another (d) 向 (b) 插入 an (e) 向 (c) 插入 an, 结果与向 (d) 插入 another 相同

下面的函数向前缀树 $t = (v', ts)$ 中插入键 s 和值 v:

$$\begin{aligned} insert\ (v', ts)\ \varnothing\ v &= (Just\ v, ts) \\ insert\ (v', ts)\ s\ v &= (v', ins\ ts) \end{aligned} \tag{6.10}$$

如果键 s 为空, 我们用 v 覆盖掉以前的值; 否则调用 ins 检查子树和它们的前缀。

$$\begin{aligned} ins\ \varnothing &= [s \mapsto (Just\ v, \varnothing)] \\ ins\ (s' \mapsto t) : ts' &= \begin{cases} match\ s\ s' : & (branch\ s\ v\ s'\ t) : ts' \\ otherwise : & (s' \mapsto t) : ins\ ts' \end{cases} \end{aligned} \tag{6.11}$$

如果节点不含任何子树, 我们创建一个值为 v 的子树, 并将 s 映射到其上; 否则, 对于每个子树映射 $s' \mapsto t$, 我们比较 s' 和 s。如果它们有公共前缀(通过 $match$ 函数检测), 则利用 $branch$ 函数分枝出子树。我们定义两个列表匹配如果它们有公共前缀:

$$\begin{aligned} match\ \varnothing\ B &= True \\ match\ A\ \varnothing &= True \\ match\ (a : as)\ (b : bs) &= a = b \end{aligned} \tag{6.12}$$

我们定义函数 $(C, A', B') = lcp\ A\ B$ 来提取列表 A 和 B 的最长公共前缀, 其中 $C + A' = A$ 且 $C + B' = B$ 成立。如果 A、B 中的任何一个为空或者它们的第一个元素不同, 则公共前缀 $C = \varnothing$; 否则, 我们递归地从子列表中提取公共前缀, 并将头部

元素附加在前：

$$\begin{array}{rcl} lcp\ \varnothing\ B & = & (\varnothing, \varnothing, B) \\ lcp\ A\ \varnothing & = & (\varnothing, A, \varnothing) \\ lcp\ (a:as)\ (b:bs) & = & \begin{cases} a \neq b: & (\varnothing, a:as, b:bs) \\ otherwise: & (a:cs, as', bs') \end{cases} \end{array} \tag{6.13}$$

其中 $(cs, as', bs') = lcp\ as\ bs$ 是递归提取的结果，函数 $branch\ A\ v\ B\ t$ 接受两个键 A、B，一个值 v 和树 t，它提取出 A、B 的最长公共前缀 C，将其映射到新分枝节点上，并设置好子树：

$$\begin{array}{l} branch\ A\ v\ B\ t = \\ \quad lcp\ A\ B = \begin{cases} (C, \varnothing, B'): & (C, (Just\ v, [B' \mapsto t])) \\ (C, A', \varnothing): & (C, insert\ t\ A'\ v) \\ (C, A', B'): & (C, (Nothing, [A' \mapsto (Just\ v, \varnothing), B' \mapsto t])) \end{cases} \end{array} \tag{6.14}$$

如果 A 是 B 的前缀，则将 A 映射到 v 所在的节点，列表的剩余部分被重新映射到分枝的唯一子树 t；如果 B 是 A 的前缀，我们递归地将剩余列表和值插入到 t 中；否则，我们创建一个值为 v 的叶子节点，将其和 t 作为分枝的两棵子树。下面的例子程序实现了 $insert$ 算法：

```
insert (PrefixTree _ ts) [] v = PrefixTree (Just v) ts
insert (PrefixTree v' ts) k v = PrefixTree v' (ins ts) where
    ins [] = [(k, leaf v)]
    ins ((k', t) : ts) | match k k' = (branch k v k' t) : ts
                       | otherwise  = (k', t) : ins ts

leaf v = PrefixTree (Just v) []

match [] _ = True
match _ [] = True
match (a:_) (b:_) = a == b

branch a v b t = case lcp a b of
  (c, [], b') → (c, PrefixTree (Just v) [(b', t)])
  (c, a', []) → (c, insert t a' v)
  (c, a', b') → (c, PrefixTree Nothing [(a', leaf v), (b', t)])

lcp [] bs = ([], [], bs)
lcp as [] = ([], as, [])
lcp (a:as) (b:bs) | a ≠ b = ([], a:as, b:bs)
                  | otherwise = (a:cs, as', bs') where
                        (cs, as', bs') = lcp as bs
```

我们也可以消除递归，用循环实现插入算法：

1: **function** INSERT(T, k, v)
2: **if** $T =$ NIL **then**

```
3:          T ← Empty-Node
4:      p ← T
5:      loop
6:          match ← FALSE
7:          for each s_i ↦ T_i in Sub-Trees(p) do
8:              if k = s_i then
9:                  Value(T_i) ← v                                  ▷ 覆盖
10:                 return T
11:             c ← LCP(k, s_i)
12:             k_1 ← k − c, k_2 ← s_i − c
13:             if c ≠ NIL then
14:                 match ← TRUE
15:                 if k_2 = NIL then                               ▷ s_i 是 k 的前缀
16:                     p ← T_i, k ← k_1
17:                     break
18:                 else                                            ▷ 新分枝
19:                     Add(Sub-Trees(p), c ↦ Branch(k_1, Leaf(v), k_2, T_i))
20:                     Delete(Sub-Trees(p), s_i ↦ T_i)
21:                     return T
22:         if not match then                                       ▷ 新叶子
23:             Add(Sub-Trees(p), k ↦ Leaf(v))
24:             break
25:     return T
```

函数 LCP 提取出两个列表的公共前缀:

```
1: function LCP(A, B)
2:     i ← 1
3:     while i ≤ |A| and i ≤ |B| and A[i] = B[i] do
4:         i ← i + 1
5:     return A[1...i − 1]
```

Branch(s_1, T_1, s_2, T_2) 中需要处理特殊情况。如果 s_1 为空,说明待插入的键是某个子树的前缀,我们将 T_2 设置为 T_1 的子树。否则,我们创建一个新的分枝节点,并将 T_1、T_2 设置为子树。

```
1: function Branch(s_1, T_1, s_2, T_2)
2:     if s_1 = NIL then
3:         Add(Sub-Trees(T_1), s_2 ↦ T_2)
4:         return T_1
5:     T ← Empty-Node
```

6: SUB-TREES$(T) \leftarrow \{s_1 \mapsto T_1, s_2 \mapsto T_2\}$
7: **return** T

虽然前缀树提高了空间利用率,但其复杂度仍然是 $O(mn)$,其中 n 是键的长度,m 是列表元素集合的大小。

6.4.3 查找

查找键 k 时,我们从根节点开始,如果 $k = \varnothing$ 为空,则返回根节点的值;否则我们检查子树的映射,找到映射 $s_i \mapsto t_i$,使得 s_i 是 k 的前缀,然后再递归地在子树 t_i 中查找 $k - s_i$。如果所有的 s_i 都不是 k 的前缀,则树中不存在要查找的键。

$$\begin{aligned} lookup\ \varnothing\ (v, ts) &= v \\ lookup\ k\ (v, ts) &= find\ ((s,t) \mapsto s \sqsubseteq k)\ ts\ = \\ & \begin{cases} Nothing: & Nothing \\ Just\ (s,t): & lookup\ (k-s)\ t \end{cases} \end{aligned} \tag{6.15}$$

其中 $A \sqsubseteq B$ 表示 A 是 B 的前缀。函数 $find$ 的定义见第一章,它在列表中查找满足指定条件的元素。下面的例子程序实现了查找算法。

```
lookup [] (PrefixTree v _) = v
lookup ks (PrefixTree v ts) =
  case find (λ(s, t) → s `isPrefixOf` ks) ts of
    Nothing → Nothing
    Just (s, t) → lookup (drop (length s) ks) t
```

前缀检查所需的时间和列表的长度成比例,$lookup$ 算法的复杂度为 $O(mn)$,其中 m 是列表元素集合的大小,n 是列表的长度。我们略过了命令式实现,将其作为本节的练习。

练习 6.4

1. 消除 $lookup$ 算法中的递归,用循环实现前缀树的查找。

6.5 Trie 和前缀树的应用

我们可以用 trie 和前缀树来解决许多有趣的问题,包括实现简单的词典,自动输入补齐,以及数字键盘输入法。与商业实现不同,本节给出的例子都是示意性的。

6.5.1 词典和自动补齐

如图6.11所示,当用户输入某些字符后,词典会搜索词库,列出候选单词。

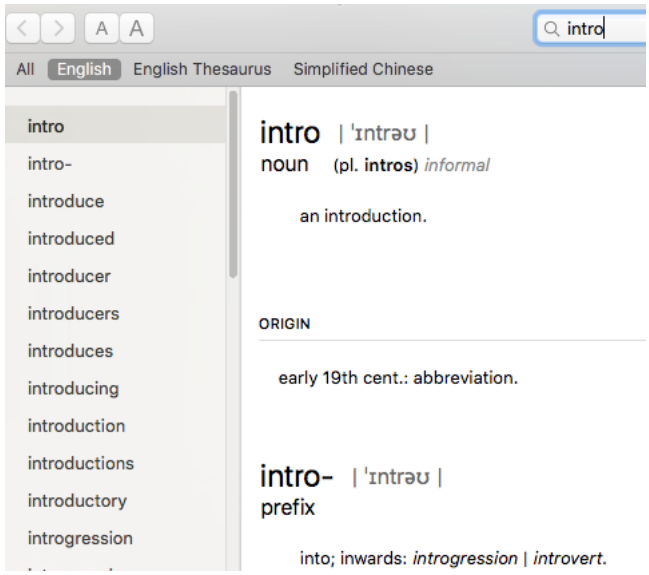


图 6.11: 词典

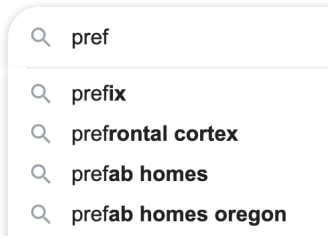


图 6.12: 带有自动补齐的输入框

词典通常存有数十万单词, 全部查找的开销很大。商业词典软件会使用多种工程方法以提高性能, 包括缓存、索引等。图6.12是一个带有自动补齐功能的输入框。当输入内容后, 会列出一些可能的候选项。它们以用户输入的内容为前缀。

这两个例子都展示了自动补齐的功能。我们可以用前缀树来实现它。简单起见, 我们在例子中限定字符为英文, 候选列表不超过 n 个条目。一个词典保存了多个键、值对, 其中键是英文单词或词组, 值是对应的意思和解释。当用户输入字符串 s 时, 我们在前缀树实现的词典中查找所有以 s 开头的键。如果 s 为空, 就扩展出所有子树直到达到 n 条结果; 否则, 我们根据匹配的子树递归查找。在支持惰性求值的环境中, 我们可以扩展出全部候选项, 然后按需取得前 n 个: $take\ n\ (startsWith\ s\ t)$, 其中 t 是前缀树。

$$\begin{aligned} startsWith\ \varnothing\ (Nothing, ts) &= enum\ ts \\ startsWith\ \varnothing\ (Just\ x, ts) &= (\varnothing, x) : enum\ ts \\ startsWith\ s\ (v, ts) &= find\ ((k,t) \mapsto s \sqsubseteq k \text{ or } k \sqsubseteq s)\ ts = \\ &\begin{cases} Nothing: & \varnothing \\ Just\ (k,t): & [(k +\!\!+ a, b) | (a,b) \in startsWith\ (s-k)\ t] \end{cases} \end{aligned} \tag{6.16}$$

给定一个列表 s, 函数 $startsWith$ 在前缀树中搜索所有以 s 为前缀的结果。如果 s 为空, 它枚举所有子树。如果根节点中的值 x 不为空, 则将 $(\varnothing, x)$ 附加到结果之前。函数 $enum\ ts$ 定义如下:

$$enum = concatMap\ (k,t) \mapsto [(k +\!\!+ a, b) | (a,b) \in startsWith\ \varnothing\ t] \tag{6.17}$$

其中 $concatMap$(也称为 $flatMap$) 是列表计算中的一个重要概念。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 融合律来实现, 以消除计算中产生的中间结果列表(见《同构——编程中的数学》第 5 章)。如果 s 不为空, 我们检查子树映射, 对于每个映射 (k,t), 如果 s 或 k 是另外一个的前缀, 我们就递归地扩展子树 t, 并将 k 附加到每个结果的键之前; 否则, 如果 s 不和任何子树的映射匹配, 则不存在以 s 为前缀的结果。下面的例子程序实现了这一算法:

```
startsWith [] (PrefixTree Nothing ts) = enum ts
startsWith [] (PrefixTree (Just v) ts) = ([], v) : enum ts
startsWith k (PrefixTree _ ts) =
  case find (λ(s, t) → s `isPrefixOf` k || k `isPrefixOf` s) ts of
    Nothing → []
    Just (s, t) → [(s ++ a, b) |
                   (a, b) ← startsWith (drop (length s) k) t]

enum = concatMap (λ(k, t) → [(k ++ a, b) | (a, b) ← startsWith [] t])
```

我们也可以用命令式的方式实现 STARTS-WITH(T, k, n)。从根节点开始, 我们循环检查每个子树映射 $k_i \mapsto T_i$。如果 k 是某个子树 T_i 的前缀, 我们就将这棵子树扩展到最多 n 条结果; 如果 k_i 是 k 的前缀, 我们就去掉前缀部分, 用新键 $k - k_i$ 在 T_i 中递归查找。

```
1: function STARTS-WITH(T, k, n)
2:     if T = NIL then
3:         return NIL
4:     s ← NIL
5:     repeat
6:         match ← FALSE
7:         for k_i ↦ T_i in SUB-TREES(T) do
8:             if k is prefix of k_i then
9:                 return EXPAND(s ++ k_i, T_i, n)
10:            if k_i is prefix of k then
11:                match ← TRUE
12:                k ← k − k_i                          ▷ 去掉前缀
13:                T ← T_i
14:                s ← s ++ k_i
15:                break
16:    until not match
17:    return NIL
```

其中函数 EXPAND(s, T, n) 从 T 中扩展出 n 个结果, 并将 s 附加在每个键的前面。我们可以用广度优先搜索方法实现它(见 14.3 节):

```
1: function EXPAND(s, T, n)
2:     R ← NIL
3:     Q ← [(s, T)]
4:     while |R| < n and Q ≠ NIL do
5:         (k, T) ← POP(Q)
6:         v ← VALUE(T)
7:         if v ≠ NIL then
8:             INSERT(R, (k, v))
9:         for k_i ↦ T_i in SUB-TREES(T) do
10:            PUSH(Q, (k ++ k_i, T_i))
```

6.5.2 数字键盘输入法

2010 年前,大多数手机上都提供一个如图6.13所示的数字键盘,称为 ITU-T 键盘。它将每个数字映射到 3 到 4 个英文字母上。如果要输入英文单词 home, 我们可以按照下面的顺序按键:

1. 按两次 4 键输入字符 h;
2. 按三次 6 键输入字符 o;

图 6.13: 手机 ITU-T 键盘

3. 按一次 6 键输入字符 m;
4. 按两次 3 键输入字符 e;

另外一种更快速的方法使用下面的按键顺序:

1. 依次按下 4、6、6、3, 候选单词 home 出现;
2. 按下‘*’号键以变换到下一个候选单词 good;
3. 按下‘*’号键再次变换到下一个候选单词 gone;
4. ……

后者称为预测式输入, 简称为 T9[25]、[26]。商业实现通常在内存和文件系统中使用多级缓存和索引。作为示例, 我们可以将单词存储在一个前缀树中来实现这种输入法。首先我们需要定义数字键盘映射:

$$\begin{aligned} M_{T9} = \{ \quad & 2 \mapsto \texttt{"abc"}, 3 \mapsto \texttt{"def"}, 4 \mapsto \texttt{"ghi"}, \\ & 5 \mapsto \texttt{"jkl"}, 6 \mapsto \texttt{"mno"}, 7 \mapsto \texttt{"pqrs"}, \\ & 8 \mapsto \texttt{"tuv"}, 9 \mapsto \texttt{"wxyz"} \quad \} \end{aligned} \tag{6.18}$$

$M_{T9}[i]$ 就给出数字 i 对应的若干字符。我们也可以定义从字符到数字的逆映射。

$$M_{T9}^{-1} = concatMap\ ((d, s) \mapsto [(c, d) | c \in s])\ M_{T9} \tag{6.19}$$

通过查找 M_{T9}^{-1}, 我们可以将字符串转换成一组按键序列。

$$digits(s) = \{M_{T9}^{-1}[c] | c \in s\} \tag{6.20}$$

对于任何不属于 [a..z] 中的字符, 我们将其映射到特殊字符'#' 上。下面的例子程序定义了上述映射:

```
mapT9 = Map.fromList [('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)
```

令 (v, ts) 是从所有候选单词构建出的前缀树。我们可以修改自动补齐算法来处理数字序列 ds。我们把每个子树映射 $(s \mapsto t) \in ts$ 中的前缀 s 转换为 $digits(s)$, 检查它是否和 ds 匹配(其中一个为另一个的前缀)。可能存在多个子树匹配 ds 的情况:

$$pfx = [(s,t)|(s \mapsto t) \in ts, digits(s) \sqsubseteq ds \ or \ ds \ \sqsubseteq digits(s)]$$

$$\begin{array}{rcl} find_{T9}\ t\ \varnothing & = & [\varnothing] \\ find_{T9}\ (v, ts)\ ds & = & concatMap\ find\ pfx \end{array} \tag{6.21}$$

对 pfx 中的每个映射 (s,t), 函数 $find$ 递归地在 t 中查找剩余数字 ds', 其中 $ds' = drop\ |s|\ ds$, 然后将 s 附加到每个候选项前面。为了防止长度超出数字个数, 我们截取前 $n = |ds|$ 个字符:

$$find\ (s,t) = [take\ n\ (s +\!\!+ s_i)|s_i \in find_{T9}\ t\ ds'] \tag{6.22}$$

下面的例子程序实现了预测输入法:

```
findT9 _ [] = [[]]
findT9 (PrefixTree _ ts) k = concatMap find pfx where
  find (s, t) = map (take (length k) ∘ (s++)) $ findT9 t (drop (length s) k)
  pfx = [(s, t) | (s, t) ← ts, let ds = digits s in
              ds `isPrefixOf` k || k `isPrefixOf` ds]
```

用命令式方法实现广度优先搜索时, 可以用一个队列 Q, 队列中的元素为三元组 $(prefix, D, t)$。每个三元组包含已搜索过的前缀 $prefix$, 尚未搜索的数字 D, 和待搜索的子树 t。队列初始的时候, 三元组包含空前缀, 全部数字, 以及前缀树的根节点。我们不断从队列中取出三元组, 检查子树的映射。对于每个映射 $(s \mapsto T')$, 我们将 s 转换成 $digits(s)$。如果 D 是它的前缀, 就找到了一个候选词。我们将 s 附加到 $prefix$ 的前面, 并记录下这一结果。如果 $digits(s)$ 是 D 的前缀, 我们需要递归在子树 T' 中搜索, 我们新建一个三元组 $(prefix +\!\!+ s, D', T')$, 其中 D' 是剩余的数字。然后将这一新三元组放回队列。

1: **function** LOOK-UP-T9(T, D)
2: $R \leftarrow$ NIL
3: **if** $T =$ NIL or $D =$ NIL **then**
4: **return** R
5: $n \leftarrow |D|$
6: $Q \leftarrow \{(\text{NIL}, D, T)\}$
7: **while** $Q \neq$ NIL **do**
8: $(prefix, D, T) \leftarrow$ POP(Q)

```
        for (s ↦ T′) ∈ SUB-TREES(T) do
            D′ ← DIGITS(s)
            if D′ ⊏ D then                          ▷ D′ 是 D 的前缀
                APPEND(R, (prefix ++ s)[1..n])      ▷ 限制长度为 n
            else if D ⊏ D′ then
                PUSH(Q, (prefix ++ s, D − D′, T′))
    return R
```

练习 6.5

1. 使用 trie 实现自动补齐和预测式输入。
2. 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?
3. 在没有惰性求值的环境中, 如何按需返回最多 n 条结果?

6.6 小结

我们从整数 trie 和整数前缀树开始, 通过整数的二进制表示, 我们复用二叉树实现了基于整数的映射(map)数据结构。接下来我们将键的类型从整数扩展到有限集元素的列表。其中一个特例就是字符串, trie 和前缀树可以用来进行文字处理。我们给出了两个应用的例子: 自动补齐和预测式输入。基数树的另外一个应用是后缀树, 它和 trie 与前缀树有着密切的关系, 是文字和 DNA 处理的有力工具。

6.7 附录: 例子程序

复用二叉树定义整数 trie:

```
data IntTrie<T> {
    IntTrie<T> left = null
    IntTrie<T> right = null
    Optional<T> value = Optional.None
}
```

下面的 *insert* 例子程序用位运算实现了奇偶测试和向右移位:

```
IntTrie<T> insert(IntTrie<T> t, Int key,
                  Optional<T> value = Optional.None) {
    if t == null then t = IntTrie<T>()
    p = t
    while key ≠ 0 {
        if key & 1 == 0 {
            p = if p.left == null then IntTrie<T>() else p.left
        } else {
            p = if p.right == null then IntTrie<T>() else p.right
```

```
        }
        key = key >> 1
    }
    p.value = Optional.of(value)
    return t
}
```

整数前缀树的定义：

```
data IntTree<T> {
    Int key
    T value
    Int prefix
    Int mask = 1
    IntTree<T> left = null
    IntTree<T> right = null

    IntTree(Int k, T v) {
        key = k, value = v, prefix = k
    }

    bool isLeaf = (left == null and right == null)

    Self replace(IntTree<T> x, IntTree<T> y) {
        if left == x then left = y else right = y
    }

    bool match(Int k) = maskbit(k, mask) == prefix
}

Int maskbit(Int x, Int mask) = x & (~(mask - 1))
```

向整数前缀树插入键、值：

```
IntTree<T> insert(IntTree<T> t, Int key, T value) {
    if t == null then return IntTree(key, value)
    node = t
    Node<T> parent = null
    while (not node.isLeaf()) and node.match(key) {
        parent = node
        node = if zero(key, node.mask) then node.left else node.right
    }
    if node.isleaf() and key == node.key {
        node.value = value
    } else {
        p = branch(node, IntTree(key, value))
        if parent == null then return p
        parent.replace(node, p)
    }
    return t
}

IntTree<T> branch(IntTree<T> t1, IntTree<T> t2) {
    var t = IntTree<T>()
```

```
    (t.prefix, t.mask) = lcp(t1.prefix, t2.prefix)
    (t.left, t.right) = if zero(t1.prefix, t.mask) then (t1, t2)
                        else (t2, t1)
    return t
}

bool zero(int x, int mask) = (x & (mask >> 1) == 0)

Int lcp(Int p1, Int p2) {
    Int diff = p1 ^ p2
    Int mask = 1
    while diff ≠ 0 {
        diff = diff >> 1
        mask = mask << 1
    }
    return (maskbit(p1, mask), mask)
}
```

trie 的定义和插入:

```
data Trie<K, V> {
  Optional<V> value = Optional.None
  Map<K, Trie<K, V>> subTrees = Map.empty()
}

Trie<K, V> insert(Trie<K, V> t, [K] key, V value) {
    if t == null then t = Trie<K, V>()
    var p = t
    for c in key {
        if p.subTrees[c] == null then p.subTrees[c] = Trie<K, V>()
        p = p.subTrees[c]
    }
    p.value = Optional.of(value)
    return t
}
```

前缀树的定义和插入:

```
data PrefixTree<K, V> {
    Optional<V> value = Optional.None
    Map<[K], PrefixTree<K, V>> subTrees = Map.empty()

    Self PrefixTree(V v) {
        value = Optional.of(v)
    }
}

PrefixTree<K, V> insert(PrefixTree<K, V> t, [K] key, V value) {
    if t == null then t = PrefixTree()
    var node = t
    loop {
        bool match = false
        for var (k, tr) in node.subtrees {
            if key == k {
```

```
                tr.value = value
                return t
            }
            prefix, k1, k2 = lcp(key, k)
            if prefix ≠ [] {
                match = true
                if k2 == [] {
                    node = tr
                    key = k1
                    break
                } else {
                    node.subtrees[prefix] = branch(k1, PrefixTree(value),
                                                   k2, tr)
                    node.subtrees.delete(k)
                    return t
                }
            }
        }
        if !match {
            node.subtrees[key] = PrefixTree(value)
            break
        }
    }
    return t
}
```

提取最长公共前缀 `lcp` 和分枝 `branch`:

```
([K], [K], [K]) lcp([K] s1, [K] s2) {
    j = 0
    while j < length(s1) and j < length(s2) and s1[j] == s2[j] {
        j = j + 1
    }
    return (s1[0..j-1], s1[j..], s2[j..])
}

PrefixTree<K, V> branch([K] key1, PrefixTree<K, V> tree1,
                        [K] key2, PrefixTree<K, V> tree2) {
    if key1 == []:
        tree1.subtrees[key2] = tree2
        return tree1
    t = PrefixTree()
    t.subtrees[key1] = tree1
    t.subtrees[key2] = tree2
    return t
}
```

枚举共同前缀的所有候选项:

```
[([K], V)] startsWith(PrefixTree<K, V> t, [K] key, Int n) {
    if t == null then return []
    [T] s = []
    repeat {
        bool match = false
```

```
        for var (k, tr) in t.subtrees {
            if key.isPrefixOf(k) {
                return expand(s ++ k, tr, n)
            } else if k.isPrefixOf(key) {
                match = true
                key = key[length(k)..]
                t = tr
                s = s ++ k
                break
            }
        }
    } until not match
    return []
}

[([K], V)] expand([K] s, PrefixTree<K, V> t, Int n) {
    [([K], V)] r = []
    var q = Queue([(s, t)])
    while length(r) < n and !q.isEmpty() {
        var (s, t) = q.pop()
        v = t.value
        if v.isPresent() then r.append((s, v.get()))
        for k, tr in t.subtrees {
            q.push((s ++ k, tr))
        }
    }
    return r
}
```

预测式输入:

```
var T9MAP={'2':"abc", '3':"def", '4':"ghi", '5':"jkl", λ
       '6':"mno", '7':"pqrs", '8':"tuv", '9':"wxyz"}

var T9RMAP = { c : d for var (d, cs) in T9MAP for var c in cs }

string digits(string w) = ''.join([T9RMAP[c] for c in w])

[string] lookupT9(PrefixTree<char, V> t, string key) {
    if t == null or key == "" then return []
    res = []
    n = length(key)
    q = Queue(("", key, t))
    while not q.isEmpty() {
        (prefix, key, t) = q.pop()
        for var (k, tr) in t.subtrees {
            ds = digits(k)
            if key.isPrefixOf(ds) {
                res.append((prefix ++ k)[:n])
            } else if ds.isPrefixOf(key) {
                q.append((prefix ++ k, key[length(k)..], tr))
            }
        }
```

```
    }
    return res
}
```

第七章　B 树

7.1　简介

上一章介绍的整数前缀树利用二叉树的边来表达信息。另一种扩展二叉树的方法是将分枝数目从 2 增加到 k。B 树是一种自平衡的 k 叉搜索树[39]。它被广泛用于计算机文件系统（基于 B+ 树，一种 B 树的扩展形势）和数据库系统。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。

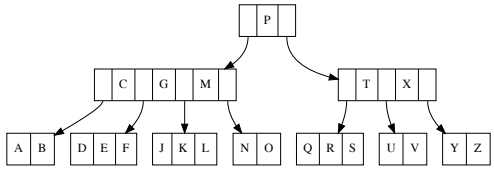


图 7.1: B 树

一棵二叉搜索树或为空，或包含一个元素 k 和左右分枝 l、r。左子树 l 中的任何元素都小于 k，并且 k 小于右子树 r 中的任何元素[1]：

$$\forall\, x \in l, y \in r \Rightarrow x < k < y \tag{7.1}$$

B 树将这一思想推广到多个分枝。一棵 B 树或为空，或包含 n 个元素和 $n+1$ 个子分枝，每个分枝也都是一棵 B 树。记这些元素为 $k_1, k_2, ..., k_n$，分枝为 $t_1, t_2, ..., t_n, t_{n+1}$，如图7.2所示。节点中的元素和分枝满足以下条件：

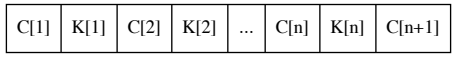


图 7.2: B 树节点

- 元素是递增的：$k_1 \leq k_2 \leq ... \leq k_n$；
- 对于任意 k_i，子树 t_i 中的所有元素都小于 k_i，并且 k_i 小于子树 t_{i+1} 中的任意元素。

[1] 严格来说，节点中可以保存键（key）和对应的值（value）。值不是必需的。简单起见，本章忽略了节点中的值，称树中保存的内容为“元素”。

$$\forall\ x_i \in t_i, i = 0, 1, ..., n\ \Rightarrow x_1 < k_1 < x_2 < k_2 < ... < x_n < k_n < x_{n+1} \tag{7.2}$$

叶子节点不包含子分枝。令元素的类型为 K，则 B 树的类型为 $BTree\ K$ 或 `BTree<K>`。此外，我们还需定义一组规则以保持 B 树平衡：

- 所有的叶子节点都有相同的深度；
- 定义整数 d，称为 B 树的最小度数，每个节点：
  - 最多含有 $2d-1$ 个元素；
  - 最少含有 $d-1$ 个元素，根节点例外。

即：

$$d - 1 \leq |keys(t)| \leq 2d - 1 \tag{7.3}$$

我们接下来证明这些规则可以保证 B 树是平衡的。

证明. 考虑一棵含有 n 个元素的 B 树，最小度数 $d \geq 2$，树的高度为 h。除根节点外，其它节点至少含有 $d-1$ 个元素。根节点至少含有一个元素。如果它有子树，则至少有两个深度为 1 的子分枝，至少有 $2d$ 个深度为 2 的子分枝，至少有 $2d^2$ 个深度为 3 的子分枝……最后，至少有 $2d^{h-1}$ 个深度为 h 的叶子节点。除根节点外，将节点个数乘以 $d-1$，B 树中存储的元素个数满足下面的不等式：

$$\begin{aligned} n &\geq 1 + (d-1)(2 + 2d + 2d^2 + ... + 2d^{h-1}) \\ &= 1 + 2(d-1)\sum_{k=0}^{h-1} d^k \\ &= 1 + 2(d-1)\frac{d^h - 1}{d-1} \\ &= 2d^h - 1 \end{aligned} \tag{7.4}$$

因此树的高度满足对数关系不等式：

$$h \leq \log_d \frac{n+1}{2} \tag{7.5}$$

□

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 $d = 2$，除根节点外的任何节点都包含 2 到 4 棵子分枝。任何红黑树本质上都可以转换为一棵 2-3-4 树。我们记度数为 d 的非空 B 树为 $(d, (ks, ts))$，其中 ks 是元素列表，ts 是子树列表。下面的例子程序定义了 B 树：

```
data BTree a = BTree [a] [BTree a]
```

空节点记为 $(\varnothing, \varnothing)$ 或 `BTree [] []`，为了避免在每个节点中都存储一份 d，我们将其和 B 树 t 组成一对值 (d, t)。

7.2 插入

插入的思路和二叉搜索树类似, 只不过需要处理多个元素和分枝。当向 B 树 t 插入元素 x 时, 我们从根节点开始, 查找这样的位置[2]:所有左侧的元素都小于 x, 而右侧的元素大于 x。如果是未满的叶子节点($|keys(t)| < 2d-1$), 就将 x 插入到此位置。否则, 这一位置会指向一棵子树 t', 我们递归地将 x 插入到 t'。

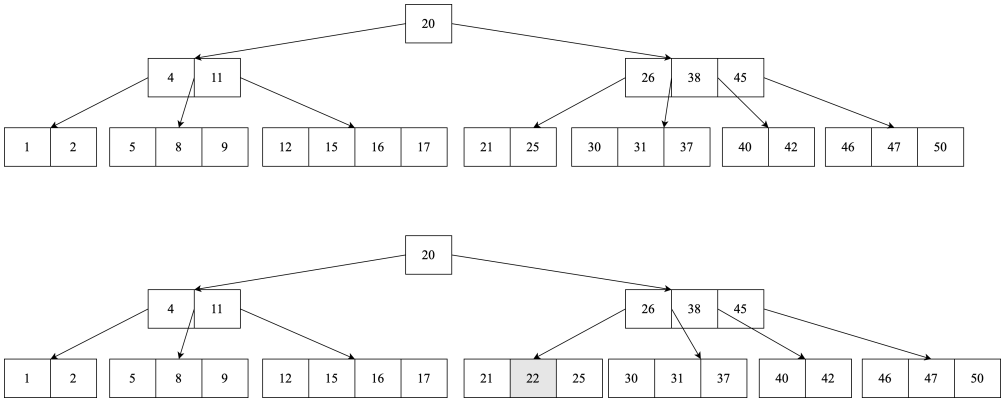


图 7.3: 将 22 插入到 2-3-4 树: $22 > 20$, 插入右子树; $22 < 26$, 插入第一棵子树。$21 < 22 < 25$, 插入到未满的叶子节点。

考虑向图7.3中的 2-3-4 树插入元素 $x = 22$。因为 $20 < 22$, 我们转向右侧的子树。它包含 26、38、45。因为 $22 < 26$, 所以接下来转向第一棵子树。它包含 21 和 25。这是一个未满的叶子节点, 将 22 插入到 21 和 25 中间。

但如果叶子节点已经含有 $2d-1$ 个元素, 插入 x 后就会因为元素过多破坏 B 树的规则。例如向图7.3插入 18 就会遇到这个问题。我们有两种解法:先插入再分拆, 和先分拆再插入。

7.2.1 先插入再分拆

我们可以将红黑树中的“先插入再修复”方法扩展到 B 树。先不考虑 B 树的平衡性, 将元素插入到适当的位置。接下来, 如果树不再平衡了, 我们自下而上对含有过多元素的节点进行分拆。首先需要定义函数, 用以判断节点是否含有过多或过少的元素。

$$\begin{cases} full\ d\ (ks, ts) & = |ks| > 2d-1 \\ low\ d\ (ks, ts) & = |ks| < d-1 \end{cases} \tag{7.6}$$

如果含有过多元素和分枝, 我们定义 $split$ 函数将其在位置 m 分拆为三部分, 如图7.4所示:

$$split\ m\ (ks, ts) = ((ks_l, ts_l), k, (ks_r, ts_r)) \tag{7.7}$$

[2]实际上, 元素只需支持小于比较和等于比较。参见练习题1。

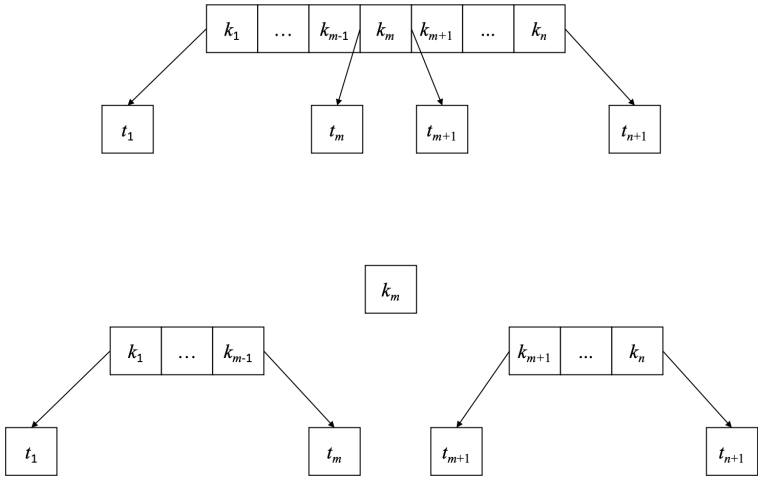


图 7.4: 在位置 m 将节点分拆为三部分。

我们使用第一章(Equation 1.55)中定义的 $splitAt$ 函数来实现:

$$\begin{cases} (ks_l, (k : ks_r)) & = splitAt\ (m-1)\ ks \\ (ts_l, ts_r) & = splitAt\ m\ ts \end{cases}$$

对称地, 我们可以定义 $unsplit$ 函数, 将三个部分合并成一个 B 树节点:

$$unsplit\ (ks_l, ts_l)\ k\ (ks_r, ts_r) = (ks_l +\!\!+ [k] +\!\!+ ks_r, ts_l +\!\!+ ts_r) \tag{7.8}$$

下面的函数先将 x 插入树 t, 然后使用 fix 修复平衡, 使其成为度数为 d 的合法 B 树:

$$insert\ x\ (d, t) = fix\ (d, ins\ t) \tag{7.9}$$

在 ins 之后, 如果根节点含有过多的元素, 函数 fix 使用 $split$ 将其分拆, 并构建新的根节点。

$$fix\ (d, t) = \begin{cases} full\ d\ t: & (d, ([k], [l, r])), \text{where } (l, k, r) = split\ d\ t \\ otherwise: & (d, t) \end{cases} \tag{7.10}$$

函数 ins 需要处理两种情况:对于叶子节点,我们可以重用第一章(Equation 1.13)定义的列表插入函数 $insert$ 来处理;否则,我们需要找到合适的位置,递归地向子树插入。为此, 我们定义函数 $partition$:

$$partition\ x\ (ks, ts) = (l, t', r) \tag{7.11}$$

其中 $l = (ks_l, ts_l)$、$r = (ks_r, ts_r)$。它进一步使用第一章(Equation 1.58)中定义的列表函数 $span$ 进行划分:

$$\begin{cases} (ks_l, ks_r) & = span\ (< x)\ ks \\ (ts_l, (t' : ts_r)) & = splitAt\ |ks_l|\ ts \end{cases}$$

这样, 所有小于 x 的元素和所在的子分枝都在左侧 l, 所有大于 x 的都在右侧 r。我们将最后一棵小于 x 的子树取出作为 t'。接下来我们递归地将 x 插入到 t' 中, 如图7.5所示。

$$\begin{array}{lcll} ins\ (ks, \varnothing) & = & (insert_L\ x\ ks, \varnothing) & \text{叶子节点, 列表插入} \\ ins\ (ks, ts) & = & balance\ d\ l\ (ins\ t')\ r & \text{其中 } (l, t', r) = partition\ x\ t \end{array} \tag{7.12}$$

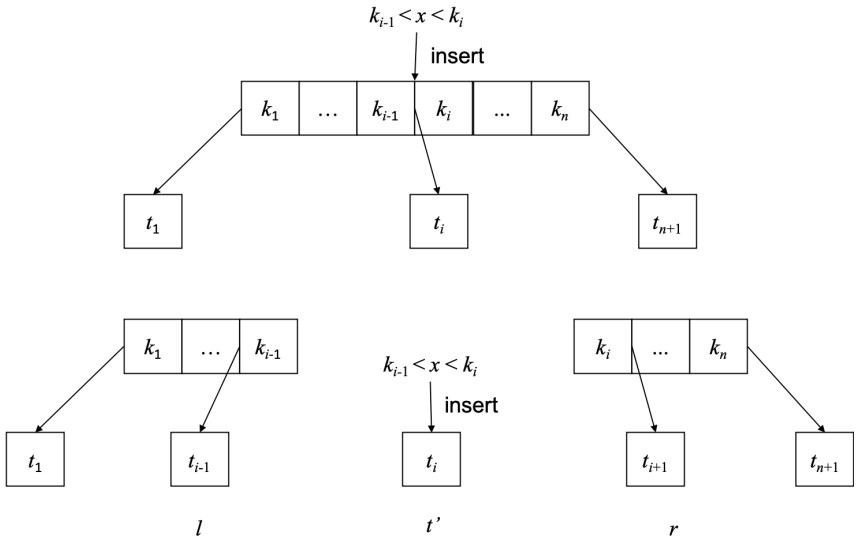


图 7.5: 用 x 划分节点

向 t' 插入 x 后, 它可能包含过多元素, 不再满足 B 树平衡条件。我们定义函数 $balance$ 递归地进行分拆修复。

$$balance\ d\ (ks_l, ts_l)\ t\ (ks_r, ts_r) = \begin{cases} full\ d\ t : fix_f \\ otherwise : (ks_l + ks_r, ts_l + [t] + ts_r) \end{cases} \tag{7.13}$$

其中 fix_f 将度数为 d 的子分枝 t 分拆为 $(t_1, k, t_2) = split\ d\ t$, 然后构建一个新的 B 树节点:

$$fix_f = (ks_l + [k] + ks_r, ts_l + [t_1, t_2] + ts_r) \tag{7.14}$$

下面的例子程序实现了 B 树的插入算法:

```
partition x (BTree ks ts) = (l, t, r) where
  l = (ks1, ts1)
  r = (ks2, ts2)
  (ks1, ks2) = span (< x) ks
```

```
  (ts1, (t:ts2)) = splitAt (length ks1) ts

split d (BTree ks ts) = (BTree ks1 ts1, k, BTree ks2 ts2) where
  (ks1, k:ks2) = splitAt (d - 1) ks
  (ts1, ts2) = splitAt d ts

insert x (d, t) = fixRoot (d, ins t) where
    ins (BTree ks []) = BTree (List.insert x ks) []
    ins t = balance d l (ins t') r where (l, t', r) = partition x t

fixRoot (d, t) | full d t  = let (t1, k, t2) = split d t in
                               (d, BTree [k] [t1, t2])
               | otherwise = (d, t)

balance d (ks1, ts1) t (ks2, ts2)
    | full d t  = fixFull
    | otherwise = BTree (ks1 ++ ks2) (ts1 ++ [t] ++ ts2)
  where
    fixFull = let (t1, k, t2) = split d t in
                BTree (ks1 ++ [k] ++ ks2) (ts1 ++ [t1, t2] ++ ts2)
```

图7.6给出了两棵 B 树的例子,它们都是依次将 “GMPXACDEJKNORSTUVYZ” 中的元素插入 B 树构造出的。

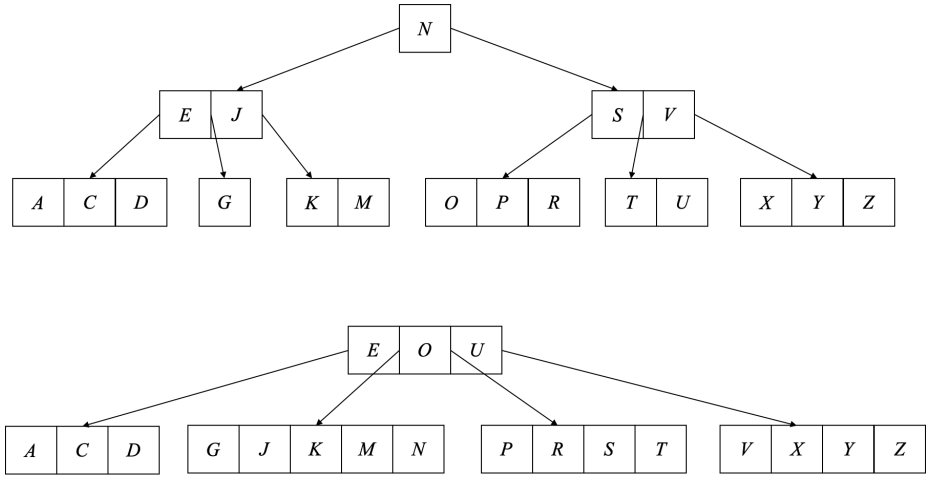


图 7.6: 依次插入 “GMPXACDEJKNORSTUVYZ”。
上:$d = 2$(2-3-4 树), 下:$d = 3$

7.2.2 先分拆再插入

第二种方法是在插入前先分拆节点以避免其含有过多元素。命令式实现常用这一方法。自顶向下递归插入时, 当遇到含有 $2d - 1$ 个元素的节点时, 我们将其分拆为三部分, 如图7.4所示。每个新节点都只含有 $d - 1$ 个元素, 即使插入元素后也仍然是合法的 B 树节点。对于节点 x, 令 $K(x)$ 表示它包含的元素, $T(x)$ 表示它包含的子分枝。

记 x 中的第 i 个元素为 $k_i(x)$, 第 j 棵子分枝为 $t_j(x)$。下面的算法在第 i 个位置对节点 z 进行分拆：

```
1: procedure SPLIT(z, i)
2:     d ← DEG(z)
3:     x ← t_i(z)
4:     y ← CREATE-NODE
5:     K(y) ← [k_{d+1}(x), k_{d+2}(x), ..., k_{2d-1}(x)]
6:     K(x) ← [k_1(x), k_2(x), ..., k_{d-1}(x)]
7:     if x is not leaf then
8:         T(y) ← [t_{d+1}(x), t_{d+2}(x), ..., t_{2d}(x)]
9:         T(x) ← [t_1(x), t_2(x), ..., t_d(x)]
10:    INSERT-AT(K(z), i, k_d(x))
11:    INSERT-AT(T(z), i + 1, y)
```

分拆节点 $x = t_i(z)$ 时, 我们将第 d 个元素 $k_d(x)$ 向上推入父节点 z。如果 z 已经满了, 推入元素后就会违反 B 树规则。为此, 我们需要从根节点起, 自顶向下沿着插入的路径进行检查, 分拆所有含有 $2d-1$ 个元素的节点。因为所有的父节点都这样被处理过, 所以可以接受推上来的元素。这一方法只需要一轮自顶向下的处理, 无需回溯。如果根节点已满, 则需要新建一个节点, 并将原来的根节点作为它的唯一子树。下面是插入算法的实现：

```
1: function INSERT(t, k)
2:     r ← t
3:     if r is full then                          ▷ 根节点已满
4:         s ← CREATE-NODE
5:         T(s) ← [r]
6:         SPLIT(s, 1)
7:         r ← s
8:     return INSERT-NONFULL(r, k)
```

其中算法 INSERT-NONFULL 假设传入的节点 r 不满。如果 r 是叶子节点, 我们按照 k 的大小将其插入到相应位置(练习3要求使用二分查找进行插入)。否则, 我们找到一个位置, 使得 $k_i(r) < k < k_{i+1}(r)$, 如果分枝 $t_i(r)$ 满了, 就进行分拆。然后继续向子分枝插入。

```
1: function INSERT-NONFULL(r, k)
2:     n ← |K(r)|
3:     if r is leaf then
4:         i ← 1
5:         while i ≤ n and k > k_i(r) do
6:             i ← i + 1
```

```
7:        INSERT-AT(K(r), i, k)
8:    else
9:        i ← n
10:       while i > 1 and k < k_i(r) do
11:           i ← i − 1
12:       if t_i(r) is full then
13:           SPLIT(r, i)
14:           if k > k_i(r) then
15:               i ← i + 1
16:       INSERT-NONFULL(t_i(r), k)
17:   return r
```

这一算法是递归的。练习2要求使用循环消除递归。图7.7给出了依次插入 “GMPXACDEJKNORSTUVYZ” 时的结果。

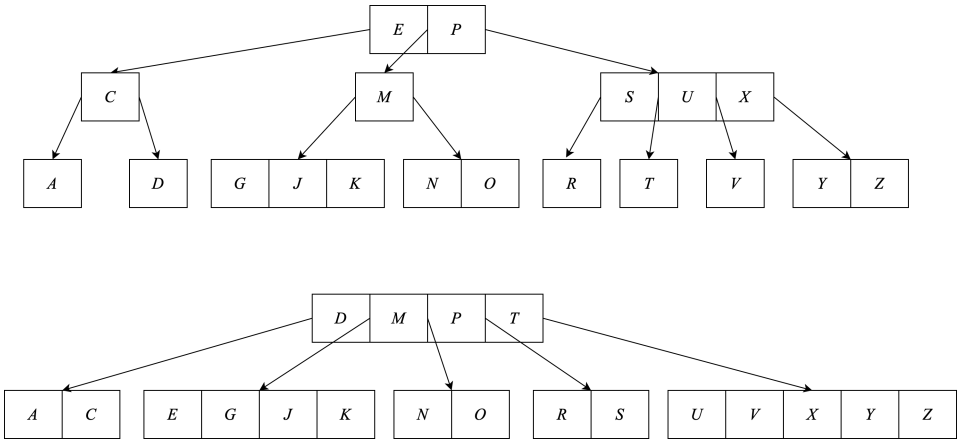


图 7.7: 依次插入 “GMPXACDEJKNORSTUVYZ”。上：$d = 2$(2-3-4 树)，下：$d = 3$

7.2.3 列表对

用列表存储元素时，我们需要从第一个元素开始，扫描列表找到插入位置。如果用数组存储，我们可以使用二分查找。可否从节点中的某个位置开始，根据元素的大小向左或向右前进呢？我们可以将 B 树节点表示为三部分：某棵子分枝 t'，它的左侧 l，右侧 r。其中左右侧都是“元素/子分枝”对 (k_i, t_i) 的列表。特别的，左侧 l 是逆序的。l 和 r 经由 t' 头对头地连接起来，组成一个如图7.8所示的马蹄形。我们可以用常数时间前后移动。

下面的例子程序用列表对定义了 B 树节点。它或者为空，或者包含三部分：左侧逆序的(元素，子分枝)列表，中间的某个子分枝，右侧的(元素、子分枝)列表。我们记非空的节点为 (l, t', r)。

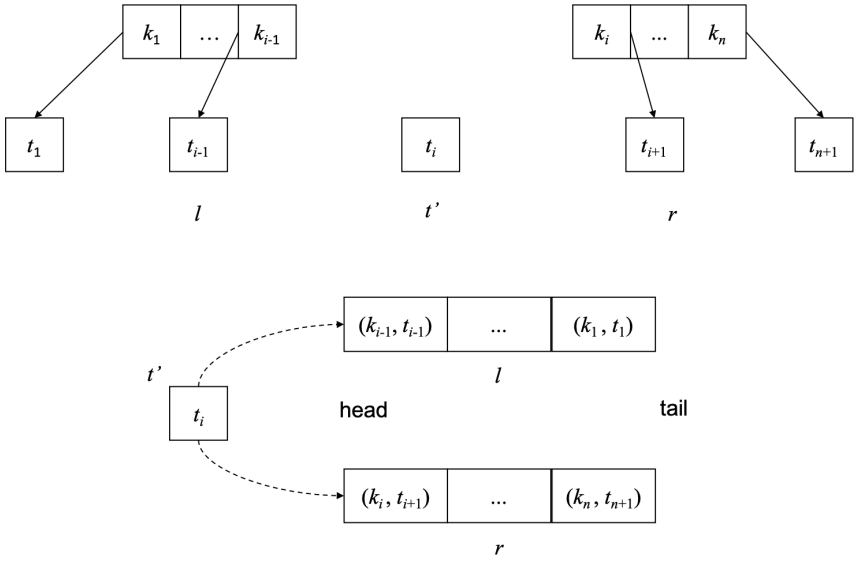


图 7.8: 将 B 树表示为某个子分枝和它两侧的一对列表

```
data BTree a = Empty
             | BTree [(a, BTree a)] (BTree a) [(a, BTree a)]
```

向右移动一步时, 我们从 r 中取出第一对值 (k, t), 组成另一对 (k, t') 置于 l 的最前面。然后用 t 替换 t'。向左移动的步骤与此对称。它们都只需要常数时间。

$$\begin{aligned} step_l\ ((k,t):l, t', r) &= (l, t, (k, t'):r) \\ step_r\ (l, t', (k,t):r) &= ((k, t'):l, t, r) \end{aligned} \tag{7.15}$$

利用左右移动, 我们可以实现划分函数 $partition\ p\ t$, 根据条件 p 把 B 树 t 分成左中右三部分: (l, m, r)。所有 l 中的分枝和 m 都满足 p, 而 r 中的分枝不满足。定义函数 $hd = fst \circ head$, 它从列表中取出第一对值 (a, b), 然后再获取 a。

$$\begin{aligned} partition\ p\ (\varnothing, m, r) &= \begin{cases} p(hd(r)): & partition\ p\ (step_r\ t) \\ otherwise: & (\varnothing, m, r) \end{cases} \\ partition\ p\ (l, m, \varnothing) &= \begin{cases} (not \circ p)(hd(l)): & partition\ p\ (step_l\ t) \\ otherwise: & (l, m, \varnothing) \end{cases} \\ partition\ p\ (l, m, r) &= \begin{cases} p(hd(l)) \text{ and } (not \circ p)(hd(r)): & (l, m, r) \\ p(hd(r)): & partition\ p\ (step_r\ t) \\ (not \circ p)(hd(l)): & partition\ p\ (step_l\ t) \end{cases} \end{aligned} \tag{7.16}$$

例如 $partition\ (< k)\ t$ 将 t 中所有不小于 k 的元素和分枝留在右侧。下面的例子程序实现了 $partition$ 函数:

```
partition p t@(BTree [] m r)
  | p (hd r) = partition p (stepR t)
  | otherwise = ([], m, r)
partition p t@(BTree l m [])
  | (not ∘ p) (hd l) = partition p (stepL t)
  | otherwise = (l, m, [])
partition p t@(BTree l m r)
  | p (hd l) && (not ∘ p) (hd r) = (l, m, r)
  | p (hd r) = partition p (stepR t)
  | (not ∘ p) (hd l) = partition p (stepL t)
```

我们也可以利用 $step_l/step_r$ 把含有过多元素的节点在位置 d 拆分。令 $n = |l|$ 表示左侧的“元素/子分枝”数量。$f^n(x)$ 表示对变量 x 重复应用函数 f 共 n 次。

$$split\ d\ t = \begin{cases} n < d: & sp(step_r^{d-n}(t)) \\ n > d: & sp(step_r^{n-d}(t)) \\ otherwise: & sp(t) \end{cases} \tag{7.17}$$

其中 sp 进行如下的分拆：

$$sp\ (l, t, (k, t') : r) = ((l, t, \varnothing), k, (\varnothing, t', r)) \tag{7.18}$$

利用 $partition$ 和 $split$，对于列表对表示的 B 树，我们可以定义出插入算法。首先我们需要修改 B 树含有过多、过少元素的判断：

$$\begin{array}{ll} full\ d\ \varnothing & = False \\ full\ d\ (l, t', r) & = |l| + |r| > 2d - 1 \end{array} \tag{7.19}$$

和

$$\begin{array}{ll} low\ d\ \varnothing & = False \\ low\ d\ (l, t', r) & = |l| + |r| < d - 1 \end{array} \tag{7.20}$$

向度数为 d 的 B 树 t 插入元素 x 时，我们首先递归地插入，然后再修复元素过多的问题：

$$insert\ x\ (d, t) = fix\ (d, ins\ t) \tag{7.21}$$

如果根节点含有过多元素，函数 fix 在位置 d 将其分拆：

$$fix\ (d, t) = \begin{cases} full\ d\ t: & (d, (\varnothing, t_1, [(k, t_2)] \text{ 其中 } (t_1, k, t_2) = split\ d\ t \\ otherwise: & (d, t) \end{cases} \tag{7.22}$$

函数 ins 需要处理 $t = \varnothing$ 和 $t \neq \varnothing$ 两种情况。对于空树，我们新建一个单独的叶子节点；否则调用 $(l, t', r) = partition\ (< x)\ t$ 定位到递归插入的位置：

$$\begin{array}{lcl} ins\ \varnothing & = & (\varnothing, \varnothing, [(x, \varnothing)]) \\ ins\ t & = & \begin{cases} t' = \varnothing: & balance\ d\ l\ \varnothing\ ((x, \varnothing) : r) \\ t' \neq \varnothing: & balance\ d\ l\ (ins\ t')\ r \end{cases} \end{array} \tag{7.23}$$

函数 $balance$ 检查子分枝 t 是否包含过多元素并进行分拆。

$$balance\ d\ l\ t\ r = \begin{cases} full\ d\ t: & fixFull \\ otherwise: & (l, t, r) \end{cases} \tag{7.24}$$

其中 $fixFull = (l, t_1, ((k, t_2) : r), (t_1, k, t_2) = split\ d\ t$。下面的例子程序实现了插入算法:

```
insert x (d, t) = fixRoot (d, ins t) where
  ins Empty = BTree [] Empty [(x, Empty)]
  ins t = let (l, t', r) = partition (< x) t in
    case t' of
      Empty → balance d l Empty ((x, Empty):r)
      _     → balance d l (ins t') r

fixRoot (d, t) | full d t = let (t1, k, t2) = split d t in
                   (d, BTree [] t1 [(k, t2)])
               | otherwise = (d, t)

balance d l t r | full d t = fixFull
                | otherwise = BTree l t r
  where
    fixFull = let (t1, k, t2) = split d t in BTree l t1 ((k, t2):r)

split d t@(BTree l _ _) | n < d = sp $ iterate stepR t !! (d - n)
                        | n > d = sp $ iterate stepL t !! (n - d)
                        | otherwise = sp t
  where
    n = length l
    sp (BTree l t ((k, t'):r)) = (BTree l t [], k, BTree [] t' r)
```

练习 7.1

1. 我们是否可以用 $\leq$ 使得 B 树含有重复元素?
2. 使用循环消除“先分拆再插入”算法中的递归。
3. 我们使用线性查找获得元素插入的位置。请使用二分查找对命令式实现进行改进。算法复杂度会提升么?

7.3 查找

我们可以将二叉搜索树的查找算法扩展到含有多个分枝的 B 树。二叉树查找只有左右两个方向,但 B 树有多个方向。考虑在 B 树 $t = (ks, ts)$ 中查找元素 k,如果 t 是叶子节点(ts 为空),则问题简化为列表查找;否则,我们用 k 将树 t 划分为三部分:$l = (ks_l, ts_l)$、t'、$r = (ks_r, ts_r)$,其中 l 和子分枝 t' 中的所有元素都小于 k,而 r 中的所有元素大于等于 k。如果 r 中的第一个元素 ks_r 等于 k,我们就找到了结果;否则我们

递归地在子分枝 t' 中查找。

$$\begin{aligned} lookup\ k\ (ks, \varnothing) &= \begin{cases} k \in ks: & Just\ (ks, \varnothing) \\ otherwise: & Nothing \end{cases} \\ lookup\ k\ (ks, ts) &= \begin{cases} Just\ k = safeHd\ ks_r: & Just\ (ks, ts) \\ otherwise: & lookup\ k\ t' \end{cases} \end{aligned} \tag{7.25}$$

其中 $((ks_l, ts_l), t', (ks_r, ts_r)) = partition\ k\ t$。函数 $safeHd$ 定义为：

$$\begin{aligned} safeHd\ [] &= Nothing \\ safeHd\ (x : xs) &= Just\ x \end{aligned}$$

下面的例子程序[3]实现了查找算法。

```
lookup k t@(BTree ks []) = if k `elem` ks then Just t else Nothing
lookup k t = if (Just k) == safeHd ks then Just t
             else lookup k t'  where
  (_, t', (ks, _)) = partition k t
```

对于列表对实现，思路是类似的。如果树不为空，我们用条件“$< k$”进行划分。然后检查右侧部分的第一个元素是否等于 k，否则再递归地进行查找：

$$\begin{aligned} lookup\ k\ \varnothing &= Nothing \\ lookup\ k\ t &= \begin{cases} Just\ k = safeFst\ (safeHd\ r): & Just\ (l, t', r) \\ otherwise: & lookup\ k\ t' \end{cases} \end{aligned} \tag{7.26}$$

其中 $(l, t', r) = partition\ (< k)\ t$ 是对非空树的划分。$safeFst$ 将函数 fst 应用到“Maybe”的值上，下面的例子程序使用了 $fmap$ 来实现：

```
lookup x Empty = Nothing
lookup x t = let (l, t', r) = partition (< x) t in
  if (Just x) == fmap fst (safeHd r) then Just (BTree l t' r)
  else lookup x t'
```

对于命令式实现，我们从根节点 r 开始，找到位置 i 使得 $k_i(r) \leq k < k_{i+1}(r)$。如果 $k_i(r) = k$，则返回节点 r 和索引 i 组成的值对；否则，我们继续在子分枝 $t_i(r)$ 中继续查找。如果 r 是叶子节点，并且 k 不在其中，则返回空结果。

1: **function** LOOK-UP(r, k)
2: **loop**
3: $i \leftarrow 1, n \leftarrow |K(r)|$
4: **while** $i \leq n$ and $k > k_i(r)$ **do**
5: $i \leftarrow i + 1$
6: **if** $i \leq n$ and $k = k_i(r)$ **then**

[3] `safeHd` 在某些程序库中以 `listToMaybe` 提供

```
7:            return (r, i)
8:        if r is leaf then
9:            return Nothing                                  ▷ k 不存在
10:       else
11:           r ← t_i(r)                                      ▷ 继续查找第 i 棵分枝
```

练习 7.2

1. 使用二分查找改进命令式查找算法。

7.4 删除

删除元素后, 节点可能因为元素不足无法满足 B 树的要求。除根节点外, 元素数不能小于 $d-1$, 其中 d 是最小度数。对称于插入算法, 我们也有两种解法: 先删除再修复、先合并再删除。

7.4.1 先删除再修复

我们首先扩展二叉搜索树的删除算法到多个分枝, 然后再修复 B 树的平衡性。算法包含两步:

$$delete\ x\ (d,t) = fix(d, del\ x\ t) \tag{7.27}$$

其中函数 del 是对多分枝扩展的删除操作。如果 t 是叶子节点, 我们从节点元素中删除 x; 否则, 我们用 x 将树划分为三部分: (l,t',r)。其中 l 和 t' 的所有元素小于 x, 而 r 中的其余元素大于等于($\geq$)x。如果 r 不为空, 我们取出其中的第一个元素 k_i, 若它等于 x(即 $k_i = x$), 我们接下来用子分枝 t' 中的最大元素 k'(即 $k' = max(t')$)取代 k_i。然后递归地从 t' 中删除 k', 如图7.9所示。否则(r 为空或 $k_i \neq x$), 我们递归地从 t' 中删除 x。

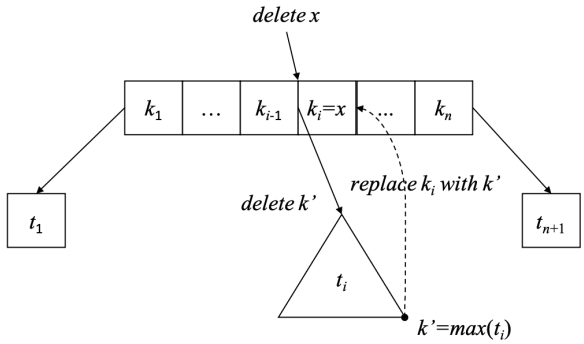


图 7.9: 用 $k' = max(t')$ 替换 k_i, 然后递归地从 t' 删除 k'

$$
\begin{aligned}
del\ x\ (ks, \varnothing) &= (delete_l\ x\ ks, \varnothing) \\
del\ x\ t &= \begin{cases} Just\ x = safeHd\ ks' : & balance\ d\ l\ (del\ k'\ t')\ (k' : (tail\ ks'), ts') \\ otherwise : & balance\ d\ l\ (del\ x\ t')\ (ks', ts') \end{cases}
\end{aligned} \tag{7.28}
$$

其中 $(l, t', (ks', ts')) = partition\ x\ t$, 是用 x 进行划分的三部分。我们可以进一步从 t' 中获得最大元素 k'。函数 max 定义如下:

$$
\begin{aligned}
max\ (ks, \varnothing) &= last\ ks \\
max\ (ks, ts) &= max\ (last\ ts)
\end{aligned} \tag{7.29}
$$

函数 $last$ 返回列表中的最后一个元素（第一章Equation 1.4)。$delete_l$ 是第一章Equation 1.16中定义的列表删除函数。$tail$ 将列表中的第一个元素去掉, 并返回剩下的元素(Equation 1.1)。我们还需要修改此前在插入算法中定义的 $balance$ 函数。如果节点中的元素太少, 就进行合并。

$$
balance\ d\ (ks_l, ts_l)\ t\ (ks_r, ts_r) = \begin{cases} full\ d\ t : fix_f \\ low\ d\ t : fix_l \\ otherwise : (ks_l + ks_r, ts_l + [t] + ts_r) \end{cases} \tag{7.30}
$$

如果 t 中的元素不足($< d - 1$), 我们调用 fix_l 与左侧 (ks_l, ts_l) 或右侧 (ks_r, ts_r) 合并(选择一个不为空的)。以左侧为例:我们从 ks_l、ts_l 中取出最后的元素 k_m、t_m。然后调用 $unsplit$(Equation 7.8)和 t 合并:$unsplit\ t_m\ k_m\ t$。构造一个含有更多元素的新分枝。最后, 我们再次调用 $balance$ 函数构造最终的 B 树。

$$
fix_l = \begin{cases} ks_l \neq \varnothing : & balance\ d\ (init\ ks_l, init\ ts_l)\ (unsplit\ t_m\ k_m\ t)\ (ks_r, ts_r) \\ ks_r \neq \varnothing : & balance\ d\ (ks_l, ts_l)\ (unsplit\ t\ k_1\ t_1)\ (tail\ ks_r, tail\ ts_r) \\ otherwise : & t \end{cases} \tag{7.31}
$$

上式最后一种情况中 $ks_l = ks_r = \varnothing$, 两侧都为空。这是一棵只有一个叶子的树, 无需进一步修复。k_1 和 t_1 分别是 ks_r 和 ts_r 中的第一个元素。最后我们修改此前插入算法中定义的 fix 函数, 加入删除的处理逻辑:

$$
\begin{aligned}
fix\ (d, (\varnothing, [t])) &= (d, t) \\
fix\ (d, t) &= \begin{cases} full\ d\ t : & (d, ([k], [l, r])), \text{其中 } (l, k, r) = split\ d\ t \\ otherwise : & (d, t) \end{cases}
\end{aligned} \tag{7.32}
$$

上式中, 我们在最前面加入一条: 如果删除后, 根节点只包含一棵子树, 我们可以缩减高度, 将此唯一的子树作为新的根。下面的例子程序实现了删除算法。

```
delete x (d, t) = fixRoot (d, del x t) where
    del x (BTree ks []) = BTree (List.delete x ks) []
```

```
    del x t = if (Just x) == safeHd ks' then
                let k' = max t' in
                   balance d l (del k' t') (k':(tail ks'), ts')
              else balance d l (del x t') r
      where
        (l, t', r@(ks', ts')) = partition x t

fixRoot (d, BTree [] [t]) = (d, t)
fixRoot (d, t) | full d t  = let (t1, k, t2) = split d t in
                               (d, BTree [k] [t1, t2])
               | otherwise = (d, t)

balance d (ks1, ts1) t (ks2, ts2)
    | full d t  = fixFull
    | low  d t  = fixLow
    | otherwise = BTree (ks1 ++ ks2) (ts1 ++ [t] ++ ts2)
  where
    fixFull = let (t1, k, t2) = split d t in
                BTree (ks1 ++ [k] ++ ks2) (ts1 ++ [t1, t2] ++ ts2)
    fixLow | not $ null ks1 = balance d (init ks1, init ts1)
                                        (unsplit (last ts1) (last ks1) t)
                                        (ks2, ts2)
           | not $ null ks2 = balance d (ks1, ts1)
                                        (unsplit t (head ks2) (head ts2))
                                        (tail ks2, tail ts2)
           | otherwise = t
```

我们将列表对 B 树的删除算法留作练习。图7.10、7.11、7.12描述了删除的例子。

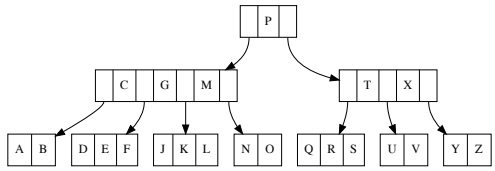


图 7.10: 删除前

7.4.2 先合并再删除

另一种方法是先把元素不足的节点合并, 然后再删除。考虑从树 t 中删除元素 x。我们先从最简单的情况入手。

情况 1:如果 x 存在于 t 的元素中, 并且 t 是叶子节点。我们可以直接将 t 中的 x 删除。如果 t 是树中的唯一节点(根), 则无需进一步修复。

情况 2:如果 x 存在于 t 的元素中, 但 t 不是叶子节点。则存在三种子情况:

情况 2a:如图7.9所示, 令 $k_i = x$ 的前驱元素为 k', 其中 $k' = max(t_i)$。如果 t_i 含有足够的元素($\geq d$), 我们用 k' 替换 k_i, 然后递归地从 t_i 中删除 k'。

情况 2b:如果 t_i 中的元素不足, 但是子分枝 t_{i+1} 含有足够的元素($\geq d$), 对称地,

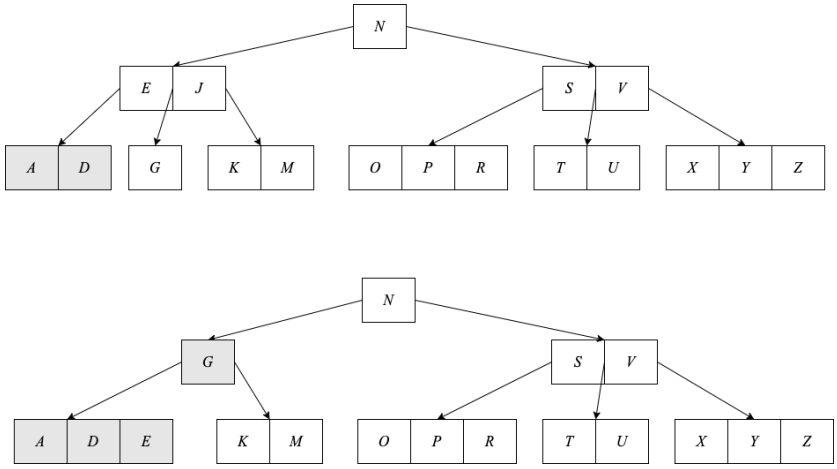


图 7.11: 删除 ‘C’, 然后删除 ‘J’

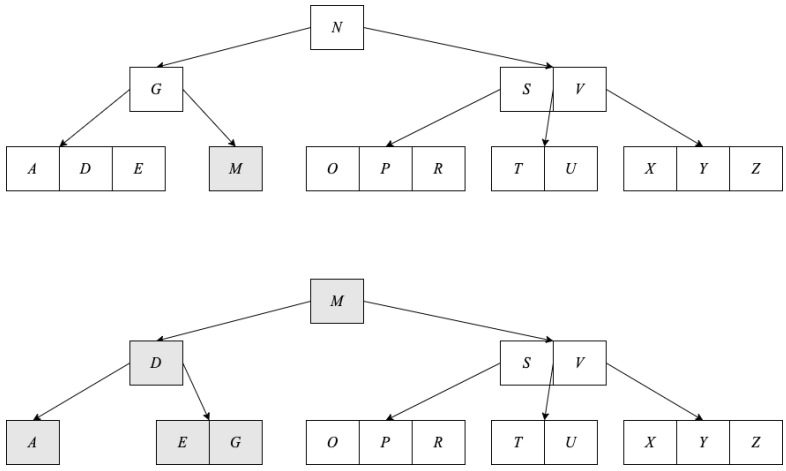


图 7.12: 删除 ‘K’, 然后删除 ‘N’

我们用后继元素 $k'' = min(t_{i+1})$ 替换 k_i, 然后递归地从 t_{i+1} 中删除 k''。如图7.13所示。

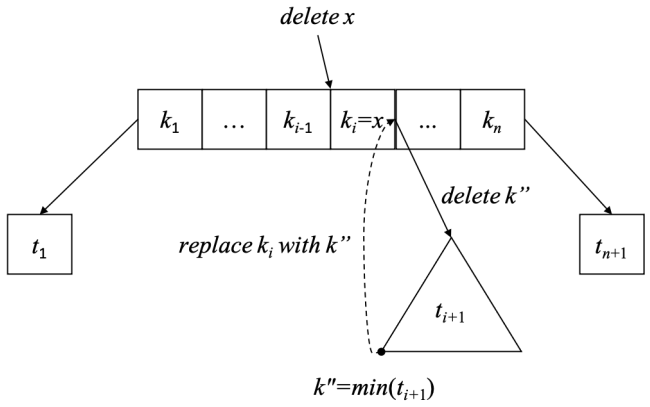


图 7.13: 用 $k'' = min(t_{i+1})$ 替换 k_i, 然后递归地从 t_{i+1} 中删除 k''

情况 2c:如果 t_i 和 t_{i+1} 的元素都不足($|t_i| = |t_{i+1}| = d-1$), 我们将 t_i、x、t_{i+1} 合并成一个新节点。它含有 $2d-1$ 个元素, 可以安全地从中删除。如图7.14所示。

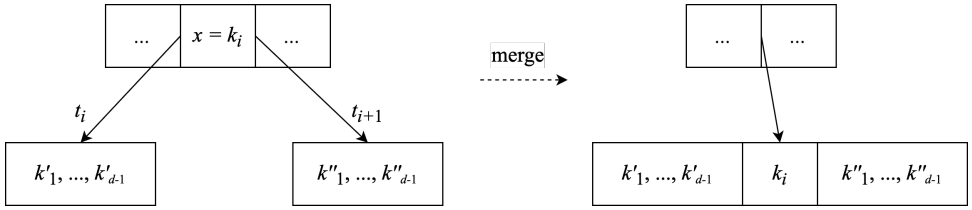


图 7.14: 先合并再删除

合并过程会将元素 k_i 推入子树。如果 t 因此变空(不再含有元素), 说明 k_i 是 t 中的唯一元素, 并且 t_i、t_{i+1} 是仅有的两棵子树。我们可以将树的高度缩减, 如图7.15所示。

图 7.15: 缩减高度

情况 3:如果 t 的元素中不包含 x, 我们需要递归地在某个子分枝 t_i 中删除 x。如果 t_i 中的元素不足, 我们需要处理两种子情况:

情况 3a:如果 t_i 的两个相邻节点 t_{i-1}、t_{i+1} 中的任何一个含有足够的元素($\geq d$),

我们将 t 中的一个元素移到 t_i, 然后将相邻节点中的一个元素向上移入 t, 将相应的子分枝移入 t_i。如图7.16所示, t_i 获得一个元素。接下来递归地从 t_i 中删除 x。

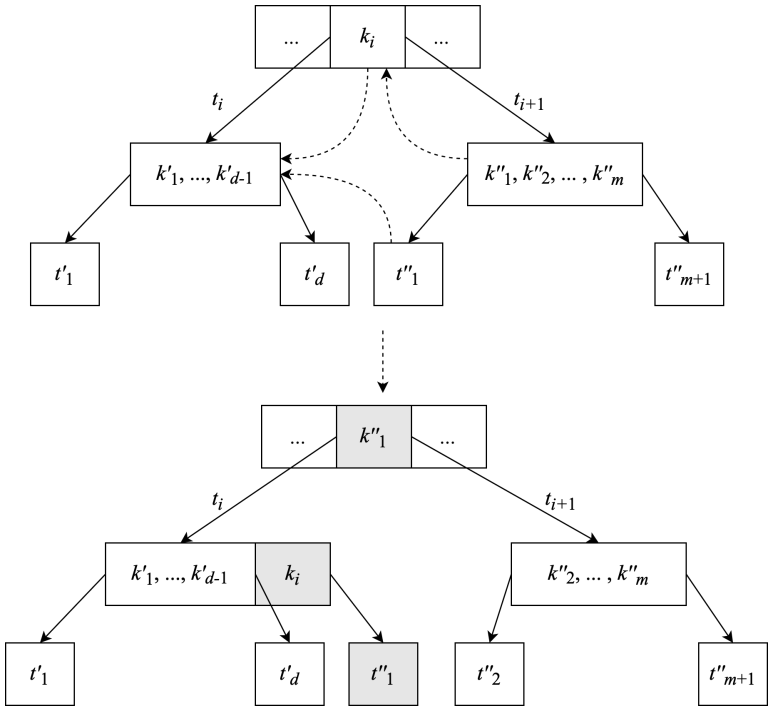


图 7.16: 从右侧移入一个元素

情况 3b:如果两个相邻节点中的元素都不足($|t_{i-1}| = |t_{i+1}| = d - 1$), 我们将 t_i, t 中的一个元素, 和任一相邻节点合并成一个新节点, 如图7.17所示。然后从中递归地删除 x。

下面的 DELETE 函数实现了先合并再删除算法:

```
1: function DELETE(t, k)
2:     if t is empty then
3:         return t
4:     i ← 1, n ← |K(t)|
5:     while i ≤ n and k > k_i(t) do
6:         i ← i + 1
7:     if k = k_i(t) then
8:         if t is leaf then                                  ▷ 情况 1
9:             REMOVE(K(t), k)
10:        else                                               ▷ 情况 2
11:            if |K(t_i(t))| ≥ d then                        ▷ 情况 2a
12:                k_i(t) ← MAX(t_i(t))
13:                DELETE(t_i(t), k_i(t))
```

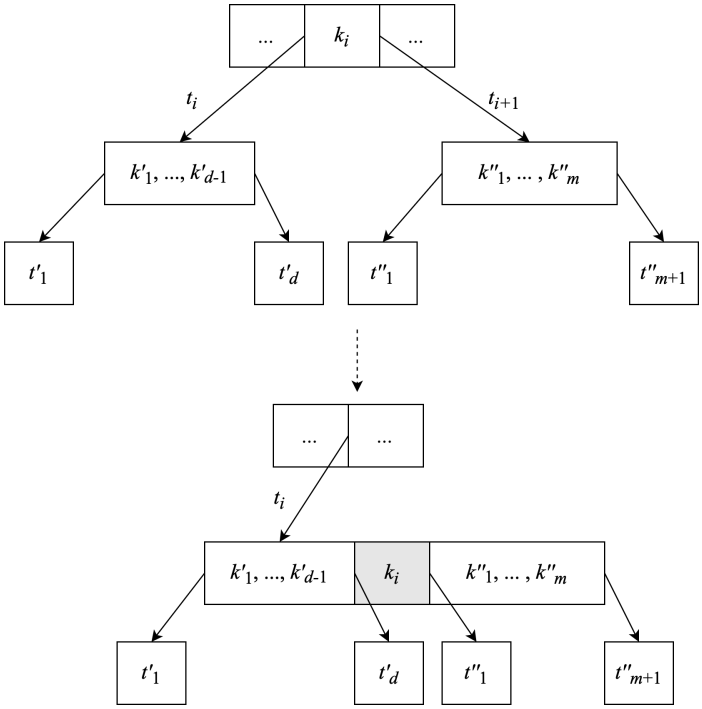


图 7.17: 合并 t_i、k、t_{i+1}

```
14:            else if |K(t_{i+1}(t))| ≥ d then                ▷ 情况 2b
15:                k_i(t) ← Min(t_{i+1}(t))
16:                Delete(t_{i+1}(t), k_i(t))
17:            else                                            ▷ 情况 2c
18:                Merge-At(t, i)
19:                Delete(t_i(t), k)
20:                if K(T) is empty then
21:                    t ← t_i(t)                              ▷ 缩减高度
22:        return t
23:    if t is not leaf then
24:        if k > k_n(t) then
25:            i ← i + 1
26:        if |K(t_i(t))| < d then                             ▷ 情况 3
27:            if i > 1 and |K(t_{i−1}(t))| ≥ d then           ▷ 情况 3a:左
28:                Insert(K(t_i(t)), k_{i−1}(t))
29:                k_{i−1}(t) ← Pop-Last(K(t_{i−1}(t)))
30:                if t_i(t) is not leaf then
31:                    Insert(T(t_i(t)), Pop-Back(T(t_{i−1}(t))))
32:            else if i ≤ n and |K(t_{i+1}(t))| ≥ d then      ▷ 情况 3a:右
```

```
33:                 APPEND(K(t_i(t)), k_i(t))
34:                 k_i(t) ← POP-FIRST(K(t_{i+1}(t)))
35:                 if t_i(t) is not leaf then
36:                     APPEND(T(t_i(t)), POP-FIRST(T(t_{i+1}(t))))
37:             else                                          ▷ 情况 3b
38:                 if i = n + 1 then
39:                     i ← i − 1
40:                 MERGE-AT(t, i)
41:         DELETE(t_i(t), k)
42:         if K(t) is empty then                             ▷ 缩减高度
43:             t ← t_1(t)
44:     return t
```

其中 MERGE-AT(t, i) 将分枝 $t_i(t)$、元素 $k_i(t)$、分枝 $t_{i+1}(t)$ 合并成一个新分枝。

```
1: procedure MERGE-AT(t, i)
2:     x ← t_i(t)
3:     y ← t_{i+1}(t)
4:     K(x) ← K(x) ++ [k_i(t)] ++ K(y)
5:     T(x) ← T(x) ++ T(y)
6:     REMOVE-AT(K(t), i)
7:     REMOVE-AT(T(t), i + 1)
```

练习 7.3

1. 我们在本节中使用了前驱子分枝中的最大元素 $k' = max(t')$ 替换要删除的元素 k, 然后递归地在 t' 中删除 k'。还有一种对称的处理方法: 用后继分枝中的最小元素来替换 k。请实现这一方法
2. 实现列表对 B 树的删除算法。

7.5 小结

B 树将二叉搜索树扩展到多个分枝, 并将分枝的数目限制在一个范围内。B 树被用来控制磁盘访问 ([4], 第 18 章)。B 树节点的分枝不会过多、过少, 平衡性得以保障。大多数的操作都和树的高度成比例, 对于含有 n 个节点的 B 树, 其性能为 $O(\lg n)$。

7.6 附录:例子程序

B 树的定义:

```
data BTree<K, Int deg> {
    [K] keys
    [BTree<K>] subStrees;
}
```

分拆节点:

```
void split(BTree<K, deg> z, Int i) {
    var d = deg
    var x = z.subTrees[i]
    var y = BTree<K, deg>()
    y.keys = x.keys[d ...]
    x.keys = x.keys[ ... d - 1]
    if not isLeaf(x) {
      y.subTrees = x.subTrees[d ... ]
      x.subTrees = x.subTrees[... d]
    }
    z.keys.insert(i, x.keys[d - 1])
    z.subTrees.insert(i + 1, y)
}

Bool isLeaf(BTree<K, deg> t) = t.subTrees == []
```

插入:

```
BTree<K, deg> insert(BTree<K, deg> tr, K key) {
    var root = tr
    if isFull(root) {
        var s = BTree<K, deg>()
        s.subTrees.insert(0, root)
        split(s, 0)
        root = s
    }
    return insertNonfull(root, key)
}
```

插入到未满的节点。

```
BTree<K, deg> insertNonfull(BTree<K, deg> tr, K key) {
    if isLeaf(tr) {
        orderedInsert(tr.keys, key)
    } else {
        Int i = length(tr.keys)
        while i > 0 and key < tr.keys[i - 1] {
            i = i - 1
        }
        if isFull(tr.subTrees[i]) {
            split(tr, i)
            if key > tr.keys[i] then i = i + 1
        }
        insertNonfull(tr.subTree[i], key)
    }
    return tr
```

```
}
```

其中 orderedInsert 按序插入元素到列表中。

```
void orderedInsert([K] lst, K x) {
    Int i = length(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
    }
}

Bool isFull(BTree<K, deg> x) = length(x.keys) ≥ 2 * deg - 1
Bool isLow(BTree<K, deg> x) = length(x.keys) ≤ deg - 1
```

迭代查找:

```
Optional<(BTree<K, deg>, Int)> lookup(BTree<K, deg> tr, K key) {
    loop {
        Int i = 0, n = length(tr.keys)
        while i < n and key > tr.keys[i] {
            i = i + 1
        }
        if i < n and key == tr.keys[i] then return Optional((tr, i))
        if isLeaf(tr) {
            return Optional.None
        } else {
            tr = tr.subTrees[i]
        }
    }
}
```

命令式先合并再删除:

```
BTree<K, deg> delete(BTree<K, deg> t, K x) {
    if empty(t.keys) then return t
    Int i = 0, n = length(t.keys)
    while i < n and x > t.keys[i] { i = i + 1 }
    if x == t.keys[i] {
        if isLeaf(t) {          // case 1
            removeAt(t.keys, i)
        } else {
            var tl = t.subtrees[i]
            var tr = t.subtrees[i + 1]
            if not low(tl) {          // case 2a
                t.keys[i] = max(tl)
                delete(tl, t.keys[i])
            } else if not low(tr) {  // case 2b
                t.keys[i] = min(tr)
                delete(tr, t.keys[i])
            } else {                  // case 2c
                mergeSubtrees(t, i)
                delete(d, tl, x)
```

```
                if empty(t.keys) then t = tl  // shrink height
            }
        return t
    }
    if not isLeaf(t) {
        if x > t.keys[n - 1] then i = i + 1
        if low(t.subtrees[i]) {
            var tl = if i == 0 then null else t.subtrees[i - 1]
            var tr = if i == n then null else t.subtrees[i + 1]
            if tl ≠ null and (not low(tl)) {   // case 3a, left
                insert(t.subtrees[i].keys, 0, t.keys[i - 1])
                t.keys[i - 1] = popLast(tl.keys)
                if not isLeaf(tl) {
                    insert(t.subtrees[i].subtrees, 0, popLast(tl.subtrees))
                }
            } else if tr ≠ null and (not low(tr)) {  // case 3a, right
                append(t.subtrees[i].keys, t.keys[i])
                t.keys[i] = popFirst(tr.keys)
                if not isLeaf(tr) {
                    append(t.subtrees[i].subtrees, popFirst(tr.subtrees))
                }
            } else {       // case 3b
                mergeSubtrees(t, if i < n then i else (i - 1))
                if i == n then i = i - 1
            }
        delete(t.subtrees[i], x)
        if empty(t.keys) then t = t.subtrees[0]    // shrink height
        }
    }
    return t
}
```

合并子分枝,获取最大、最小元素。

```
void mergeSubtrees(BTree<K, deg>, Int i) {
    t.subtrees[i].keys += [t.keys[i]] + t.subtrees[i + 1].keys
    t.subtrees[i].subtrees += t.subtrees[i + 1].subtrees
    removeAt(t.keys, i)
    removeAt(t.subtrees, i + 1)
}

K max(BTree<K, deg> t) {
    while not empty(t.subtrees) {
        t = last(t.subtrees)
    }
    return last(t.keys)
}

K min(BTree<K, deg> t) {
    while not empty(t.subtrees) {
        t = t.subtrees[0]
    }
    return t.keys[0]
```

```
}
```

第八章　二叉堆

8.1　定义

堆是一种常见的数据结构, 可以解决很多实际问题, 包括排序、带有优先级的调度、实现图算法等[40]。堆有多种实现, 最常见的一种通过数组来表示二叉树[4], 进而实现堆。许多程序库中的堆都是这样实现堆的。由 R.W. Floyd 给出的高效堆排序算法也利用了这个实现[41] [42]。堆本身的定义是抽象的, 除数组外, 它也可以由其它数据结构来实现。本章中, 我们介绍那些由二叉树实现的堆, 包括左偏堆 (Leftist heap)、斜堆 (Skew heap)、伸展堆 (splay heap)[3]。一个堆或为空, 或存有若干可比较大小的元素。它满足一条性质并定义了三种操作:

1. **性质**: 堆顶总保存着最小元素;
2. **弹出操作**移除堆顶元素, 并保持堆的性质: 新的堆顶元素仍是剩余中最小的;
3. **插入操作**将新元素加入堆中, 并保持堆的性质;
4. **其它操作** (如合并两个堆) 也保持堆的性质。

由于元素可比较, 我们也可以令堆顶总保存最大元素。我们称顶部保存最小元素的堆为**小顶堆**, 顶部保存最大元素的堆为**大顶堆**。可以用树来实现堆。将最小 (或最大) 元素置于根节点。获取"堆顶"元素时, 可以直接返回根节点中的数据。执行"弹出"操作时, 将根节点删除, 然后从子节点重新构建树。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。

8.2　由数组实现的隐式二叉堆

第一种实现称为"隐式二叉树"。它用数组来表示一棵完全二叉树。所谓完全二叉树, 是一种"几乎"满的二叉树。深度为 k 的满二叉树含有 $2^k - 1$ 个节点。如果将每个节点从上到下, 从左向右编号为 1,2, ..., $2^k - 1$, 则完全二叉树中编号为 i 的节点和满二叉树中编号为 i 的节点在树中的位置相同。完全二叉树的叶子节点仅出现在最下面一行和倒数第二行。图8.1给出了一棵完全二叉树和相应的数组表示形式。由于二叉树是完全的, 对数组中第 i 个元素代表的节点, 它的父节点定位到第 $\lfloor i/2 \rfloor$ 个元素; 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）。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）：

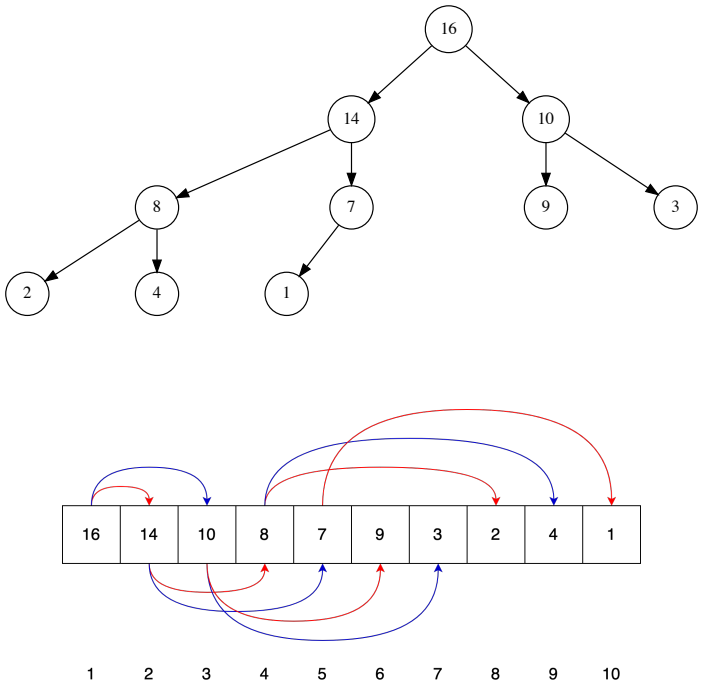


图 8.1: 完全二叉树到数组的映射

$$\begin{cases} parent(i) & = \lfloor \frac{i}{2} \rfloor \\ left(i) & = 2i \\ right(i) & = 2i + 1 \end{cases} \tag{8.1}$$

父节点和子树的访问可以通过位运算实现，见本章附录中的例子。

8.2.1 堆调整

堆调整[1]是维护堆性质的过程，使得堆顶元素为最小（或最大）。由于堆背后的数据模型是二叉树，我们可以利用树的递归特性，获得一个增强的堆性质：使得每棵子树的根节点都是最小（或最大）的。也就是说任何子树都代表一个子堆。简单起见，我们考虑小顶堆。对于用数组表示的二叉堆，任给数组下标 i，我们检查 i 对应的所有子节点的值是否都不小于 $(\geq)$ 它。如果不满足则交换，使得父节点总保存最小值[4]，并对以 i 为根的所有子树递归重复这个过程。如下算法定义了堆调整过程：

1: **function** HEAPIFY(A, i)

[1] Heapify，也译作堆化。

```
    n ← |A|
    loop
        s ← i                                          ▷ s 指向最小
        l ← LEFT(i), r ← RIGHT(i)
        if l ≤ n 且 A[l] < A[i] then
            s ← l
        if r ≤ n 且 A[r] < A[s] then
            s ← r
        if s ≠ i then
            EXCHANGE A[i] ↔ A[s]
            i ← s
        else
            return
```

对于数组 A 和索引 i, 堆性质要求 $A[i]$ 的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, 然后自顶向下检查并调整堆, 使得所有子树都满足堆性质。HEAPIFY 的时间复杂度为 $O(\lg n)$, 其中 n 是元素个数。这是因为算法中的循环次数和完全二叉树的高度成正比。图8.2描述了 HEAPIFY 从索引 2 开始, 按照小顶堆调整数组 [1, 13, 7, 3, 10, 12, 14, 15, 9, 16] 的步骤。数组最终变换为 [1, 3, 7,9, 10, 12, 14, 15, 13, 16]。

8.2.2 构造堆

使用 HEAPIFY, 我们可以从任意数组构造出堆。观察完全二叉树各层的节点数目: $1, 2, 4, 8, ...$ 都是 2 的整数次幂, 唯一例外是最后一层。由于树不一定满, 最后一层最多含有 2^{p-1} 个节点, 其中 p 是使得 $2^p - 1 \geq n$ 的最小整数, n 是数组的长度。HEAPIFY 对叶子节点不起作用, 因为叶子节点都已经满足堆性质了。我们跳过叶子节点, 从第一个分支节点开始不断向上执行 HEAPIFY。显然第一个分支节点的索引不大于 $\lfloor n/2 \rfloor$。我们可以设计出如下的堆构造算法:

```
1: function BUILD-HEAP(A)
2:     n ← |A|
3:     for i ← ⌊n/2⌋ down to 1 do
4:         HEAPIFY(A, i)
```

虽然 HEAPIFY 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:

$$S = n(\frac{1}{4} + 2\frac{1}{8} + 3\frac{1}{16} + ...) \tag{8.2}$$

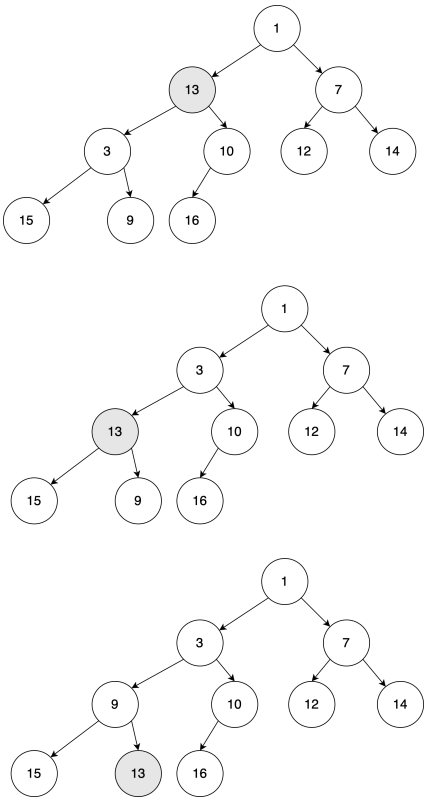


图 8.2: 堆调整。第一步：13、3、10 中最小值为 3, 交换 $3 \leftrightarrow 13$；第二步：13、15、9 中最小值为 9, 交换 $13 \leftrightarrow 9$；第三步：13 为叶子节点, 调整结束。

将两侧都乘以 2：

$$2S = n(\frac{1}{2} + 2\frac{1}{4} + 3\frac{1}{8} + ...) \tag{8.3}$$

用式(8.3)减去式(8.2)，我们有：

$$\begin{aligned} 2S - S &= n[\frac{1}{2} + (2\frac{1}{4} - \frac{1}{4}) + (3\frac{1}{8} - 2\frac{1}{8}) + ...] && \text{错开第一项两两相减} \\ S &= n[\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + ...] && \text{等比级数和} \\ &= n \end{aligned}$$

图8.3描述了从数组 $[4, 1, 3, 2, 16, 9, 10, 14, 8, 7]$ 构造小顶堆的过程。黑色表示执行 HEAPIFY 的目标节点；灰色表示进行交换的节点。

8.2.3 堆的基本操作

堆的基本操作包括获取顶部，弹出顶部，寻找最小(或最大)的前 k 个元素，减小小顶堆中某一元素(或增大大顶堆中某一元素)，以及插入新元素。在用二叉树实现的堆中，根节点保存了顶部元素，它对应数组的第一个值：

```
1: function TOP(A)
2:     return A[1]
```

弹出堆顶

弹出堆顶后，数组剩余元素顺次向前移动。这相当于删除了二叉树的根节点，剩余部分不再保持一棵树状的结构。为了避免这种情况，我们可以将待移除的数组头部和末尾元素交换，然后将数组的长度减一，这相当于删除叶子节点而非根节点。最后再用 HEAPIFY 恢复堆性质：

```
1: function POP(A)
2:     x ← A[1], n ← |A|
3:     EXCHANGE A[1] ↔ A[n]
4:     REMOVE(A, n)
5:     if A is not empty then
6:         HEAPIFY(A, 1)
7:     return x
```

从数组的末尾删除元素仅需常数时间，这样弹出操作的时间复杂度就取决于 HEAPIFY，为 $O(\lg n)$。

Top-k

连续使用 pop，可以找出一组元素中的前 k 个最小(或最大)：

```
1: function TOP-K(A, k)
```

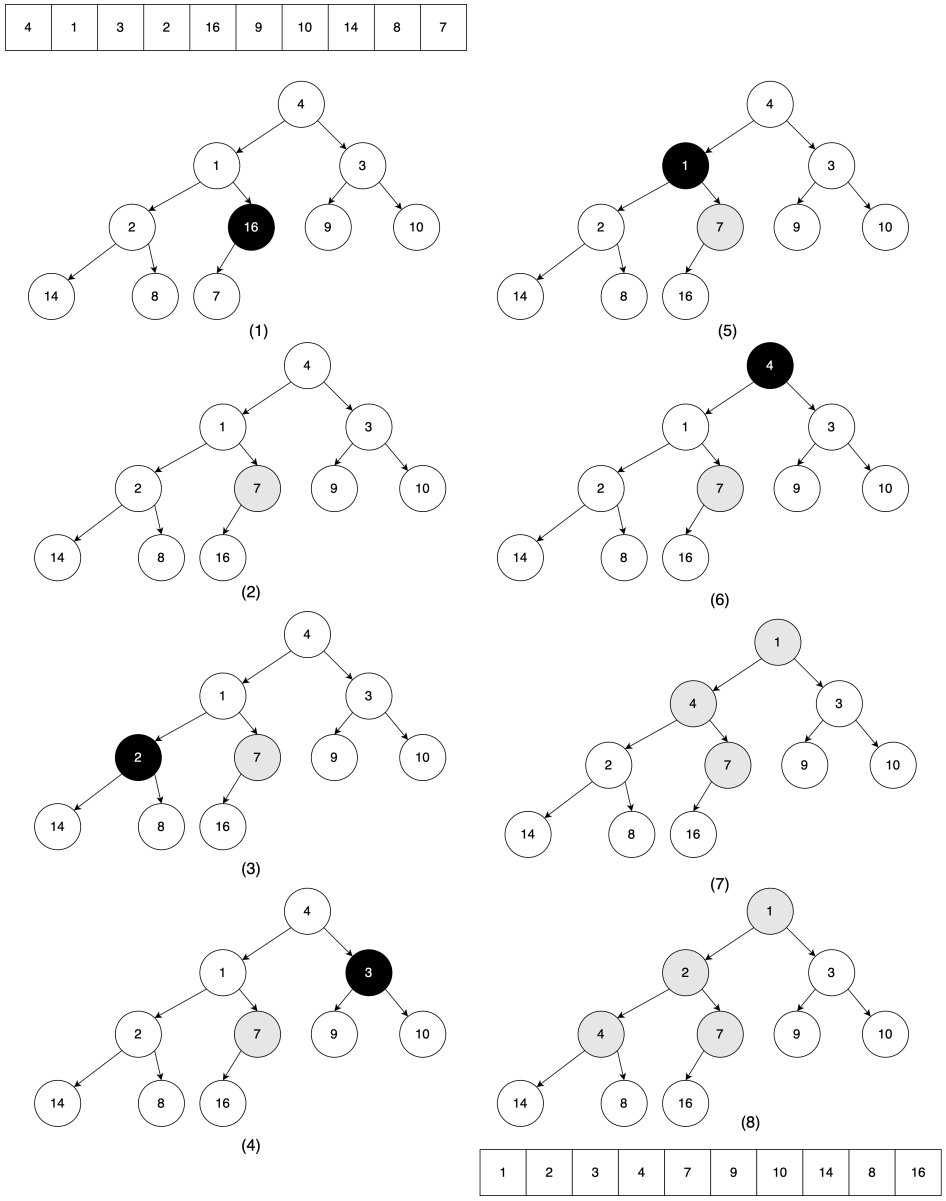


图 8.3: 构造堆。(1)检查 16, 它大于子节点 7;(2)交换 16 ↔ 7;(3)检查 2, 它比 14, 8 都小;(4)检查 3, 它比 9 和 10 都小;(5)检查 1, 它比 2 和 7 都小;(6)检查 4, 1 比 4 和 3 更小;(7)交换 4 ↔ 1;(8)交换 4 ↔ 2, 结束。

```
    R ← [ ]
    BUILD-HEAP(A)
    loop MIN(k, |A|) times                    ▷ k 超出长度则截断
        APPEND(R, POP(A))
    return R
```

提升优先级

堆的一个应用是实现带有优先级的任务调度, 称为“优先级队列”。将若干带有优先级的任务放入堆中, 每次从堆顶取出优先级最高的任务执行。为了尽早执行堆中的某个任务, 可以提升它的优先级。对于小顶堆, 这意味着减小某个元素的值, 如图8.4所示。

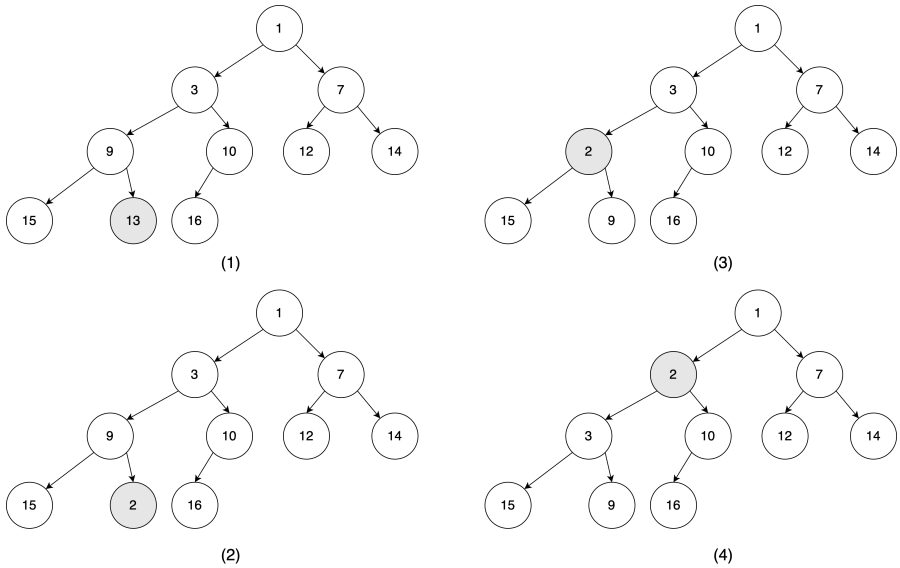


图 8.4: 将 13 减小为 2。2 先与 9 交换, 然后再与 3 交换。

减小小顶堆中的某个元素时, 可能会破坏堆性质。对于数组表示的二叉堆, 令修改的元素索引为 i, 下面的算法自底向上恢复堆性质, 其时间复杂度为 $O(\lg n)$。

```
function HEAP-FIX(A, i)
    while i > 1 and A[i] < A[ PARENT(i) ] do
        EXCHANGE A[i] ↔ A[ PARENT(i) ]
        i ← PARENT(i)
```

插入

可以利用 HEAP-FIX 来实现插入[4]。以小顶堆为例, 先向数组末尾添加新元素 k, 再使用 HEAP-FIX 调整:

```
1: function PUSH(A, k)
2:     APPEND(A, k)
3:     HEAP-FIX(A, |A|)
```

8.2.4 堆排序

可以利用堆实现排序。以小顶堆为例, 从待排序元素构建一个堆, 不断从堆顶获取最小元素, 就获得了升序结果。若待排序的元素有 n 个, 构建堆的时间复杂度是 $O(n)$。由于弹出操作的复杂度为 $O(\lg n)$, 并且共执行了 n 次。因此总体时间复杂度为 $O(n \lg n)$。由于我们使用了另外一个列表存放排序结果, 因此空间复杂度为 $O(n)$。

```
1: function HEAP-SORT(A)
2:     R ← [ ]
3:     BUILD-HEAP(A)
4:     while A ≠ [ ] do
5:         APPEND(R, POP(A))
6:     return R
```

Robert. W. Floyd 给出了另一种实现。思路是构建一个大顶堆。接下来, 将堆顶的最大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。我们将堆的大小减一, 然后执行 HEAPIFY 恢复堆的性质。重复这一过程, 直到堆中仅剩下一个元素。这一算法省去了额外的空间。

```
1: function HEAP-SORT(A)
2:     BUILD-MAX-HEAP(A)
3:     n ← |A|
4:     while n > 1 do
5:         EXCHANGE A[1] ↔ A[n]
6:         n ← n − 1
7:         HEAPIFY(A[1...n], 1)
```

练习 8.1

1. 考虑另外一种实现原地堆排序的想法: 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。这一方法正确么?

   ```
   1: function HEAP-SORT(A)
   2:     BUILD-HEAP(A)
   3:     for i = 1 to n − 1 do
   4:         HEAPIFY(A[i...n], 1)
   ```

2. 类似地, 可以通过自左向右执行 k 遍 HEAPIFY 来实现 top-k 算法么?

```
1: function TOP-K(A, k)
2:     BUILD-HEAP(A)
3:     n ← |A|
4:     for i ← 1 to min(k, n) do
5:         HEAPIFY(A[i...n], 1)
```

8.3 左偏堆和斜堆

考虑不用数组而用显式的二叉树实现堆。当弹出堆顶元素后, 剩余部分是左右两棵子树, 它们都是堆, 如图8.5所示。我们如何将它们再次合并成一个堆呢? 为保持堆性质, 新的根节点必须是剩余元素中最小的。我们可以先给出两个特殊情况下的结果:

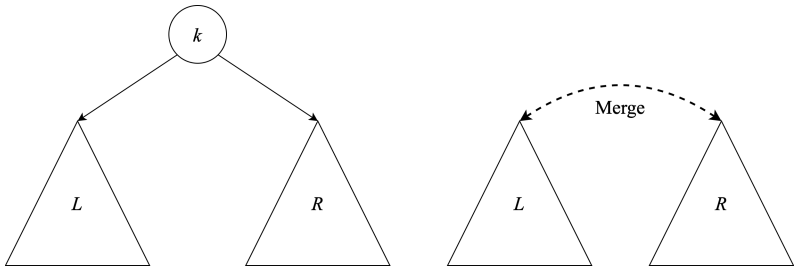


图 8.5: 弹出堆顶(删除根节点)后, 合并左右子树形成一个堆。

$$\begin{aligned} merge(\varnothing, R) &= R \\ merge(L, \varnothing) &= L \\ merge(L, R) &= ? \end{aligned}$$

左右子树都不为空时, 由于它们也都是堆, 各自的根节点都保存了最小的元素。我们可以比较两棵子树的根, 选择较小的一个作为合并后的根。令 $L = (A, x, B)$、$R = (A', y, B')$, 其中 A、A'、B、B' 都是子树。如果 $x < y$, x 就是新的根。我们可以保留 A, 然后递归地合并 B 和 R; 或者保留 B, 递归地合并 A 和 R。新的堆可以为 $(merge(A, R), x, B)$ 或 $(A, x, merge(B, R))$ 之一。两个都可以, 为了简单, 我们可以总选择右子树进行合并。这样的堆称为**左偏堆**(*Leftist* heap)。

8.3.1 左偏堆

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]。树中每个节点都定义了一个秩, 也称作 S 值。节点的秩是到最近的 NIL 节点的距离。而 NIL 节点的秩等于 0。如图8.6所示, 距离根节点 4 最近的叶子节点为 8, 所以根节点的秩为 2。节点 6 和 8 都是叶子, 它们的秩为 1。虽然节点 5 的左子树不为空, 但是

它的右子树为空, 因此秩等于 1。使用秩, 我们可以定义合并策略如下, 记左右子树的秩分别为 r_l、r_r:

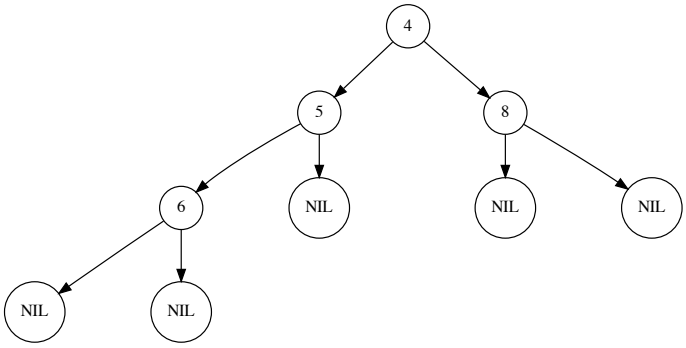


图 8.6: $rank(4) = 2$、$rank(6) = rank(8) = rank(5) = 1$

1. 总是合并右子树;
2. 若 $r_l < r_r$, 就交换左右子树。

我们称这样的合并策略为“左偏性质”。概括来说, 在一棵左偏树中, 到某个 NIL 的最短距离总在右侧。左偏树总是趋向不平衡, 但是它可以维护一条重要的性质:

定理 8.3.1. 若一棵左偏树 T 包含 n 个节点, 从根节点到达最右侧 NIL 的路径上最多含有 $\lfloor \log(n+1) \rfloor$ 个节点。

我们这里省略了证明[44],[45]。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。我们可以在二叉树的基础上增加一个秩来定义左偏树。记非空的左偏树为 (r, L, k, R), 分别表示秩、左子树、元素值、右子树:

```
data LHeap a = E -- 空
             | Node Int (LHeap a) a (LHeap a)
```

定义 $rank$ 函数返回节点的秩。

$$\begin{aligned} rank\ \varnothing &= 0 \\ rank\ (r, L, k, R) &= r \end{aligned} \tag{8.4}$$

合并

为了实现合并, 我们定义 $make$ 函数比较左右子树的秩, 并交换子树保持左偏性质。

$$make(A, k, B) = \begin{cases} rank(A) < rank(B): & (rank(A)+1, B, k, A) \\ \text{否则}: & (rank(B)+1, A, k, B) \end{cases} \tag{8.5}$$

传入两棵子树 A、B 和元素 k。若 A 的秩较小，则用 B 作为左子树、A 作为右子树。新节点的秩为 $rank(A)+1$；否则，若 B 的秩较小，就用 A 作为左子树，B 作为右子树。新节点的秩为 $rank(B)+1$。给定两个左偏堆 H_1 和 H_2，它们不空时分别记为 (r_1, L_1, K_1, R_1)、(r_2, L_2, k_2, R_2)，下面的函数定义了合并操作：

$$\begin{aligned} merge\ \varnothing\ H_2 &= H_2 \\ merge\ H_1\ \varnothing &= H_1 \\ merge\ H_1\ H_2 &= \begin{cases} k_1 < k_2: & make(L_1, k_1, merge\ R_1\ H_2) \\ \text{否则}: & make(L_2, k_2, merge\ H_1\ R_2) \end{cases} \end{aligned} \tag{8.6}$$

$merge$ 总在右子树上进行递归，左偏性质得以保持。这样就保证了算法的复杂度为 $O(\lg n)$。回顾上节，使用数组实现的堆在大多数情况下性能很好，并且和计算机的高速缓存技术配合良好。但是合并操作的时间复杂度却为线性时间 $O(n)$。我们需要将两个数组连接起来，然后重新构建堆[50]。

1: **function** MERGE-HEAP(A, B)
2: $C \leftarrow$ CONCAT(A, B)
3: BUILD-HEAP(C)

使用 $merge$ 函数，可以实现基本的堆操作。

弹出顶部

我们可以在 $O(1)$ 时间内获取根节点中的堆顶元素（设若树不为空）：

$$top\ (r, L, k, R) = k \tag{8.7}$$

弹出顶部后，我们将左右子树合并为一个新堆。弹出的时间复杂度和 $merge$ 相同，也是 $O(\lg n)$。

$$pop\ (r, L, k, R) = merge\ L\ R \tag{8.8}$$

插入

插入新元素 k 时，可以从 k 构造出只有一个叶子节点的树，然后将它和合并到待插入的树中：

$$insert\ k\ H = merge\ (1, \varnothing, k, \varnothing)\ H \tag{8.9}$$

或写成克里化的形式 $insert\ k\ = merge\ (1, \varnothing, k, \varnothing)$。这样插入的时间复杂度也和合并相同，为 $O(\lg n)$。我们可以将一个列表中的元素依次插入，从列表构造左偏堆。图8.7给出了一个构造左偏堆的例子。

$$build\ L = fold_r\ insert\ \varnothing\ L \tag{8.10}$$

对应的克里化形式为：$build = fold_r\ insert\ \varnothing$

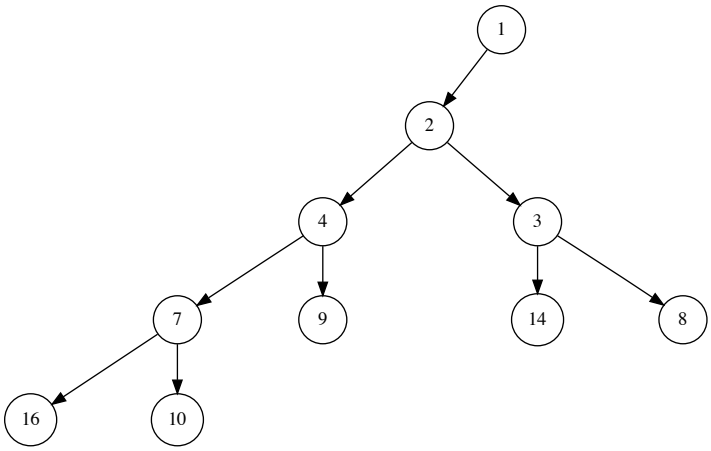


图 8.7: 从列表 $[9, 4, 16, 7, 10, 2, 14, 3, 8, 1]$ 构造左偏堆

堆排序

给定一个序列, 将它转换成一个左偏堆, 然后不断取出堆顶的最小元素就可以实现排序:

$$sort = heapSort \circ build \tag{8.11}$$

其中:

$$\begin{array}{rcl} heapSort\ [\,] & = & [\,] \\ heapSort\ H & = & (top\ H) : (heapSort\ (pop\ H)) \end{array} \tag{8.12}$$

弹出需要对数时间, 并且被调用了 n 次, 因此排序的总复杂度为 $O(n \lg n)$。

8.3.2 斜堆

左偏堆在某些情况下会产生不平衡的结构, 如图8.8所示。斜堆(skew heap)是一种自调整堆, 它简化了左偏堆的实现并提高了平衡性[46]、[47]。构造左偏堆时, 若左侧的秩小于右侧, 则交换左右子树。但是这一策略不能很好处理某一分支含有一个 NIL 子节点的情况: 不管树有多大, 它的秩总为 1。斜堆简化了合并策略: 每次都交换左右子树。

斜堆是由斜树(skew tree)实现的堆。斜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斜树。它不保存秩, 可以直接复用二叉树的定义。树或者为空, 或记为 (L, k, R)。

```
data SHeap a = E -- 空
             | Node (SHeap a) a (SHeap a)
```

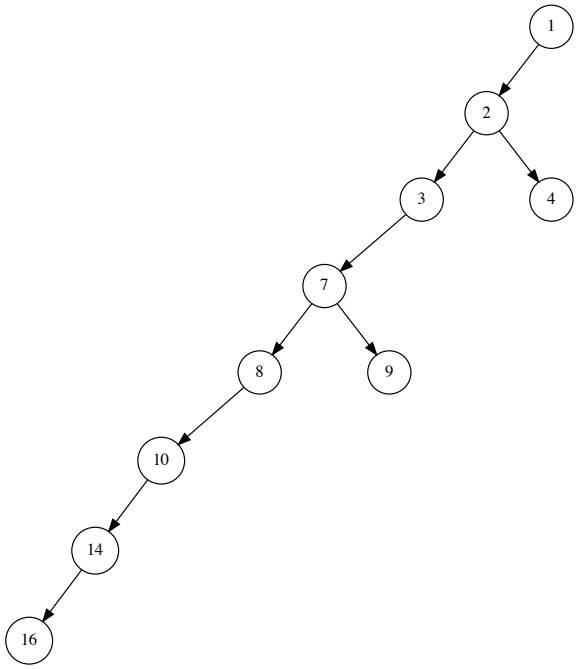


图 8.8: 从序列 $[16, 14, 10, 8, 7, 9, 3, 2, 4, 1]$ 构造的左偏堆

合并

当合并两棵非空斜树时, 先比较根节点, 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。令两棵非空子树为: $H_1 = (L_1, k_1, R_1)$、$H_2 = (L_2, k_2, R_2)$。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 $(merge(R_1, H_2), k_1, L_1)$。

$$\begin{aligned} merge\ \varnothing\ H_2 &= H_2 \\ merge\ H_1\ \varnothing &= H_1 \\ merge\ H_1\ H_2 &= \begin{cases} k_1 < k_2: & (merge(R_1, H_2), k_1, L_1) \\ \text{否则}: & (merge(H_1, R_2), k_2, L_2) \end{cases} \end{aligned} \tag{8.13}$$

其他的操作, 包括插入, 获取和弹出顶部都和左偏树一样通过合并来实现。即使用斜堆处理已序序列, 结果仍然是一棵较平衡的二叉树, 如图8.9所示。

8.4 伸展堆

左偏堆和斜堆是直接用二叉树实现的堆。如果将二叉树换成二叉搜索树, 则最小(或最大)元素就不再位于根节点。我们需要 $O(\lg n)$ 时间来获取最小(或最大)元素。

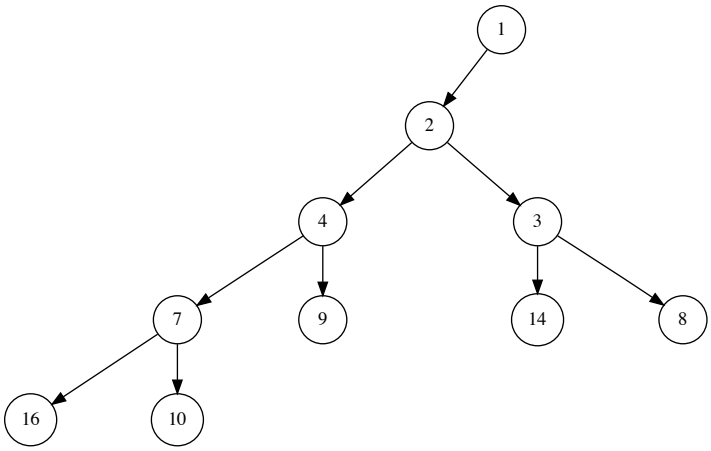


图 8.9: 用已序序列 $[1, 2, ..., 10]$ 构造的斜树

如果二叉搜索树不平衡, 性能会下降。最坏情况下会退化为 $O(n)$。尽管可以用红黑树来保持平衡, 伸展树提供了一种轻量级的实现方法, 使得树不断趋向平衡。伸展树采用类似于缓存的策略, 它不断将正在访问的节点向根旋转, 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, 经过若干次伸展后, 树会变得逐渐平衡。大多数伸展树操作的均摊性能是 $O(\lg n)$ 的。Daniel Dominic Sleator 和 Robert Endre Tarjan 在 1985 年最早引入了伸展树[48] [49]。

8.4.1 伸展操作

有两种方法可以实现伸展操作。第一种利用模式匹配, 但需要处理较多的情况; 第二种具备统一的形式, 但是实现较为复杂。记正在访问的节点元素为 x, 它的父节点元素为 p, 如果存在祖父节点, 其元素记为 g。伸展操作分为三个步骤, 每个步骤有两个对称的情况, 我们以每步中的一种情况举例说明, 如图8.10所示。

1. zig-zig 步骤, 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。

2. zig-zag 步骤, 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。

3. zig 步骤, 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。

共有 6 种不同的情况。记非空二叉树为 $T = (L, k, R)$, 当访问树中的元素 y 时,

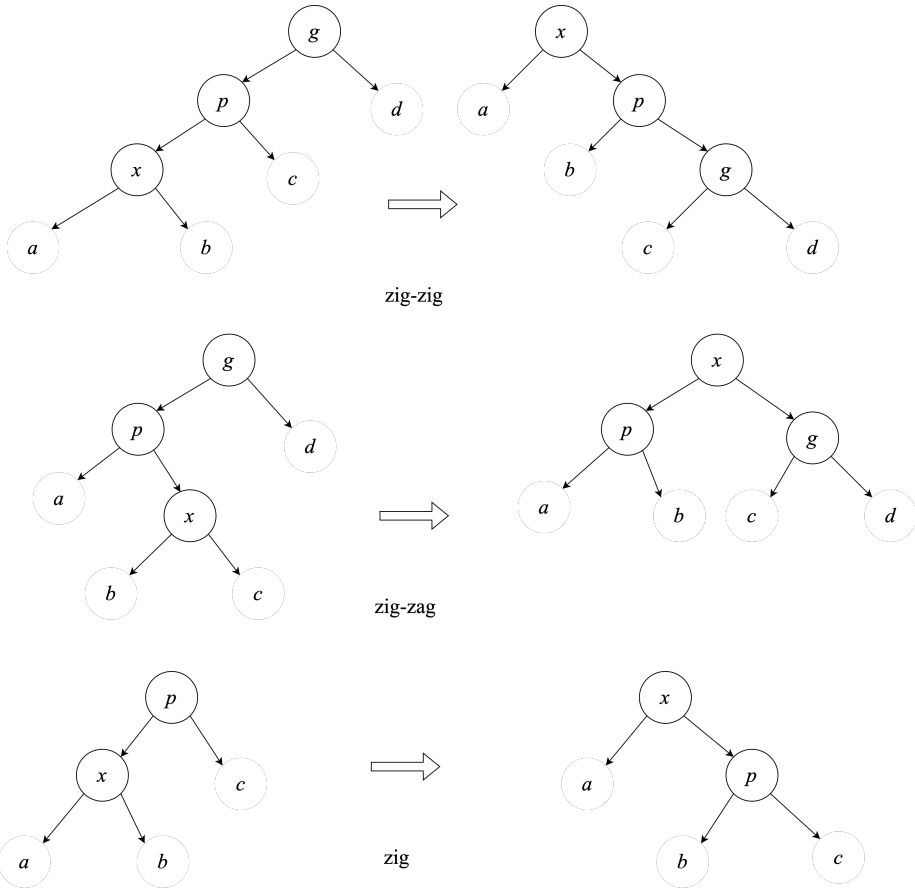


图 8.10: zig-zig: x 和 p 都是左子树或都是右子树, x 成为根节点。zig-zag: x 和 p 一棵是左子树另一棵是右子树, x 成为根节点, p 和 g 变成了兄弟节点。zig: p 是根节点, 旋转后 x 变为根节点。

伸展操作定义如下：

$$
\begin{array}{rcll}
splay\ (((a,x,b),p,c),g,d)\ y & = & \begin{cases} x=y: & (a,x,(b,p,(c,g,d))) \\ \text{否则}: & T \end{cases} & \text{zig-zig} \\
splay\ (a,g,(b,p,(c,x,d)))\ y & = & \begin{cases} x=y: & (((a,g,b),p,c),x,d) \\ \text{否则}: & T \end{cases} & \text{zig-zig 对称} \\
splay\ (a,p,(b,x,c),g,d)\ y & = & \begin{cases} x=y: & ((a,p,b),x,(c,g,d)) \\ \text{否则}: & T \end{cases} & \text{zig-zag} \\
splay\ (a,g,((b,x,c),p,d))\ y & = & \begin{cases} x=y: & ((a,g,b),x,(c,p,d)) \\ \text{否则}: & T \end{cases} & \text{zig-zag 对称} \\
splay\ ((a,x,b),p,c)\ y & = & \begin{cases} x=y: & (a,x,(b,p,c)) \\ \text{否则}: & T \end{cases} & \text{zig} \\
splay\ (a,p,(b,x,c))\ y & = & \begin{cases} x=y: & ((a,p,b),x,c) \\ \text{否则}: & T \end{cases} & \text{zig 对称} \\
splay\ T\ y & = & T & \text{其它}
\end{array}
\tag{8.14}
$$

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。每次插入新元素时，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元素和根节点，如果待插入的元素较小，就将其递归插入左子树，然后执行伸展操作；否则插入右子树，再执行伸展操作。

$$
\begin{array}{rcl}
insert\ \varnothing\ y & = & (\varnothing, y, \varnothing) \\
insert\ (L,x,R)\ y & = & \begin{cases} y<x: & splay\ ((insert\ L\ y),x,R)\ y \\ \text{否则}: & splay\ (L,x,(insert\ R\ y))\ y \end{cases}
\end{array}
\tag{8.15}
$$

图8.11给出了逐一插入 $[1,2,...,10]$ 的结果。伸展树产生了较平衡的结果。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，就旋转节点。当访问节点 x 的时候，如果连续向左侧或者右侧前进两次，我们将树 T 分割成两部分：L 和 R，其中 L 中的所有元素小于 x，R 中的元素都大于 x。然后以 x 为根，

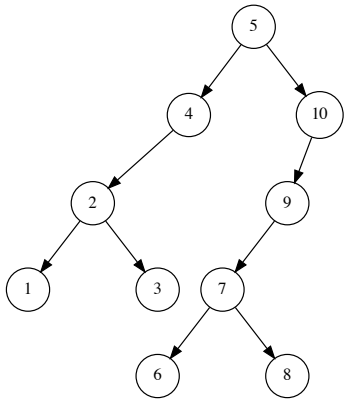


图 8.11: 从 $[1, 2, ..., 10]$ 产生的伸展树。

L、R 为左右子树构建一棵新树。分割过程递归地对子树进行伸展操作。

$$
\begin{array}{rcl}
partition\ \varnothing\ y & = & (\varnothing, \varnothing) \\
partition\ (L, x, R)\ y & = & \\
\left\{\begin{array}{ll}
x < y & \left\{\begin{array}{ll}
R = \varnothing & (T, \varnothing) \\
R = (L', x', R') & \left\{\begin{array}{ll}
x' < y & (((L, x, L'), x', A), B) \\
 & 其中: (A, B) = partition\ R'\ y \\
否则 & ((L, x, A), (B, x', R')) \\
 & 其中: (A, B) = partition\ L'\ y
\end{array}\right.
\end{array}\right. \\
否则 & \left\{\begin{array}{ll}
L = \varnothing & (\varnothing, T) \\
L = (L', x', R') & \left\{\begin{array}{ll}
y < x' & (A, (L', x', (R', x, R))) \\
 & 其中: (A, B) = partition\ L'\ y \\
否则 & ((L', x', A), (B, x, R)) \\
 & 其中: (A, B) = partition\ R'\ y
\end{array}\right.
\end{array}\right.
\end{array}\right.
\end{array}
\tag{8.16}
$$

$partition$ 接受一棵树 T 和一个基准值 y。如果树为空, 结果为一对空的左右子树。否则, 记树为 (L, x, R), 我们比较基准值 y 和根节点的值 x。如果 $x < y$, 分为两种子情况。(1) R 为空, 根据二叉搜索树的性质, 所有的元素都小于 y, 结果为 $(T, \varnothing)$。(2) 否则, 令 $R = (L', x', R')$, 若 $x' < y$, 我们递归地用基准值分割 R', 将 R' 中所有小于 y 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 $((L, x, L'), x', A)$, 另一棵为 B。若 $x' > y$, 我们递归地用 y 分割 L' 得到结果 (A, B)。最终的结果为一对树, 一棵是 (L, x, A), 另一棵是 (B, x', R')。当 $y < x$ 时, 情况是对称的。

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。

$$insert\ T\ k = (L, k, R), \text{其中:} (L, R) = partition\ T\ k \tag{8.17}$$

8.4.2 弹出顶部

由于伸展树本质上是二叉搜索树，最小的元素位于最左侧的节点。我们不断向左遍历以获取顶部元素。记非空的树为 $T = (L, k, R)$，top 函数可以定义如下：

$$\begin{aligned} top\ (\varnothing, k, R) &= k \\ top\ (L, k, R) &= top\ L \end{aligned} \tag{8.18}$$

这实际上就是二叉搜索树的 min 函数。弹出顶部时，需要将最小元素删除。同时每当连续向左访问两次，就执行一次伸展操作。

$$\begin{aligned} pop\ (\varnothing, k, R) &= R \\ pop\ ((\varnothing, k', R'), k, R) &= (R', k, R) \\ pop\ ((L', k', R'), k, R) &= (pop\ L', k', (R', k, R)) \end{aligned} \tag{8.19}$$

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。伸展树在平衡时，堆顶操作的时间复杂度是 $O(\lg n)$。

8.4.3 合并

通过使用 $partition$，我们可以实现 $O(\lg n)$ 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。

$$\begin{aligned} merge\ \varnothing\ T &= T \\ merge\ (L, x, R)\ T &= ((merge\ L\ L')\ x\ (merge\ R\ R')) \end{aligned} \tag{8.20}$$

其中：

$$(L', R') = partition\ T\ x$$

如果第一个堆为空，结果为另一个堆。否则，记第一个堆为 (L, x, R)，用 x 为基准值分割 T 得到结果 (L', R')，其中 L' 包含 T 中所有小于 x 的元素，而 R' 包含其余元素。接下来递归地将 L 和 L' 合并为新的左子树，将 R 和 R' 合并为右子树。

8.5 小结

本章中，我们介绍了通用的二叉堆概念。只要满足堆的性质，可以使用各种形式的二叉树来实现。用数组作为存储结构适于命令式实现，它将一棵完全二叉树映射为

数组。方便进行随机访问。我们还介绍了直接用二叉树实现的堆, 适于函数式实现。大部分操作的时间复杂度可以达到 $O(\lg n)$, 有些操作的分摊时间复杂度是 $O(1)$ 的。Okasaki 在[3] 中给出了详细分析。一个自然的想法是将二叉树扩展到多叉树, 这样就会得到其它数据结构如二项式堆、斐波那契堆和配对堆。参见第十章。

练习 8.2

1. 用命令式的方式实现左偏堆、斜堆、伸展堆。

8.6 附录:例子程序

在数组表示的完全二叉树中, 利用位运算访问父节点和子树, 索引从 0 开始:

```
Int parent(Int i) = ((i + 1) >> 1) - 1

Int left(Int i) = (i << 1) + 1

Int right(Int i) = (i + 1) << 1
```

堆调整, 将元素间的比较运算抽象为参数:

```
void heapify([K] a, Int i, Less<K> lt) {
    Int l, r, m
    Int n = length(a)
    loop {
        m = i
        l = left(i)
        r = right(i)
        if l < n and lt(a[l], a[i]) then m = l
        if r < n and lt(a[r], a[m]) then m = r
        if m ≠ i {
            swap(a, i, m);
            i = m
        } else {
            break
        }
    }
}
```

从数组构造堆:

```
void buildHeap([K] a, Less<K> lt) {
    Int n = length(a)
    for Int i = (n-1) / 2 downto 0 {
        heapify(a, i, lt)
    }
}
```

弹出堆顶:

```
K pop([K] a, Less<K> lt) {
    var n = length(a)
    t = a[n]
    swap(a, 0, n - 1)
    remove(a, n - 1)
    if a ≠ [] then heapify(a, 0, lt)
    return t
}
```

寻找 top-k:

```
[K] topk([K] a, Int k, Less<K> lt) {
    buildHeap(a, lt)
    [K] r = []
    loop min(k, length(a)) {
        append(r, pop(a, lt))
    }
    return r
}
```

减小堆中某元素的值:

```
void decreaseKey([K] a, Int i, K k, Less<K> lt) {
    if lt(k, a[i]) {
        a[i] = k
        heapFix(a, i, lt)
    }
}

void heapFix([K] a, Int i, Less<K> lt) {
    while i > 0 and lt(a[i], a[parent(i)]) {
        swap(a, i, parent(i))
        i = parent(i)
    }
}
```

堆插入:

```
void push([K] a, K k, less<K> lt) {
    append(a, k)
    heapFix(a, length(a) - 1, lt)
}
```

堆排序:

```
void heapSort([K] a, less<K> lt) {
    buildHeap(a, not ∘ lt)
    n = length(a)
    while n > 1 {
        swap(a, 0, n - 1)
        n = n - 1
        heapify(a[0 .. (n - 1)], 0, not ∘ lt)
    }
}
```

左偏堆合并:

```
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
```

斜堆合并:

```
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'
```

伸展操作:

```
-- 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
```

伸展堆的插入:

```
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
```

伸展树的分割:

```
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)
```

伸展树合并：

```
merge E t = t
merge (Node l x r) t = Node (merge l l') x (merge r r')
    where (l', r') = partition t x
```

第九章　选择排序

9.1　简介

本章介绍另一种直观的排序方法——选择排序。它在性能上不如快速排序和归并排序等分治算法。我们给出选择排序性能的简要分析，并且从不同的角度加以改进，最终演进到堆排序，从而达到基于比较的排序算法性能上限 $O(n \lg n)$。选择排序的思想可以在日常生活中找到。观察孩子们吃葡萄时，会发现两种类型的吃法：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。第一种孩子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选择排序。选择排序的算法描述为：

1. 如果序列为空，排序结果也为空；
2. 否则，找到最小的元素，将其附加到结果的后面。

这一算法产生升序结果。如果每次选择最大的元素，则结果是降序的。我们可以用抽象的比较操作实现排序。

$$\begin{array}{rcl} sort\ [\,] & = & [\,] \\ sort\ A & = & m : sort\ (A - [m]) \quad \text{其中 } m = min\ A \end{array} \tag{9.1}$$

其中 $A - [m]$ 是从序列 A 中去除元素 m 后的剩余部分。对应的命令式描述为：

```
1: function SORT(A)
2:     X ← [ ]
3:     while A ≠ [ ] do
4:         x ← MIN(A)
5:         DEL(A, x)
6:         APPEND(X, x)
7:     return X
```

图9.1描述了选择排序的过程。作为改进，我们可以在 A 中进行原地排序，去掉列表 X。将最小的元素保存在 $A[1]$，将次小的元素保存在 $A[2]$……。可以通过交换位置实现这一改进：当找到第 i 小的元素后，将它和 $A[i]$ 交换。

```
1: function SORT(A)
```

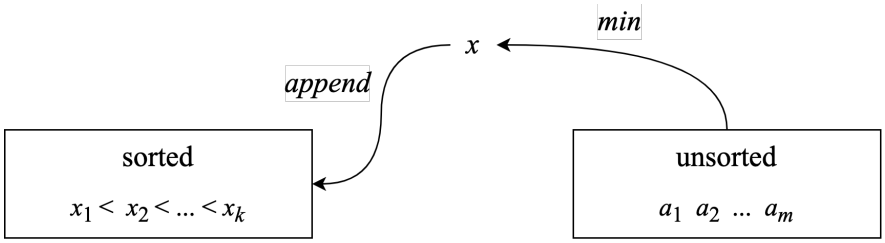


图 9.1: 左侧部分为已序元素, 不断从剩余部分选择最小元素附加在左侧尾部

```
2:    for i ← 1 to |A| do
3:        m ← Min-At(A, i)
4:        Exchange A[i] ↔ A[m]
```

令 $A = [a_1, a_2, ..., a_n]$, 当处理第 i 个元素时, $[a_1, a_2, ..., a_{i-1}]$ 都已排序。我们找到 $[a_i, a_{i+1}, ..., a_n]$ 中的最小元素, 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9.2描述了这一思路。

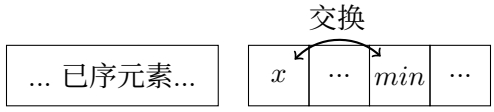


图 9.2: 左侧部分为已序元素, 不断从剩余部分找到最小的交换到正确位置

9.2 查找最小元素

我们可以用比较——交换方法在一组元素中寻找最小值。将元素编号为 $1, 2, ..., n$。比较编号为 $1, 2$ 的两个元素, 选择较小的留下与编号为 3 的元素比较……重复这一步骤直到第 n 号元素。这一方法适合处理数组。

```
1: function Min-At(A, i)
2:     m ← i
3:     for i ← m + 1 to |A| do
4:         if A[i] < A[m] then
5:             m ← i
6:     return m
```

Min-At 查找片断 $A[i...]$ 中最小元素的位置 m。令 m 指向第一个元素 $A[i]$, 并逐一检查元素 $A[i+1], A[i+2], ...$。

也可以用递归的方法在在一组元素 L 中查找最小值。如果 L 只含有一个元素, 它就是最小元素。否则从 L 中取出一个元素 x, 然后在剩余部分中递归找到最小元素 y。

x、y 中较小的一个就是最终的最小元素。

$$\begin{aligned} min\ [x] &= (x, [\,]) \\ min\ (x : xs) &= \begin{cases} x < y: & (x, xs),\ \text{其中}:\ (y, ys) = min\ xs \\ \text{否则}: & (y,\ x : ys) \end{cases} \end{aligned} \tag{9.2}$$

可以进一步用尾递归优化。将全部元素分成两组：A、B。开始时 A 为空([ ])，B 包含全部元素。我们从 B 中任选两个元素比较，将较大的放入 A，而留下较小的记为 m。此后不断从 B 中任取一个元素，和 m 对比直到 B 变成空。这时，m 就是最小的元素。在任何时候有不变关系：$L = A + [m] + B$，其中 $a \leq m \leq b, a \in A, b \in B$。

$$min\ (x : xs) = min'\ [\,]\ x\ xs \tag{9.3}$$

其中：

$$\begin{aligned} min'\ as\ m\ [\,] &= (m, A) \\ min'\ as\ m\ (b : bs) &= \begin{cases} b < m: & min'\ (m : as)\ b\ bs \\ \text{否则}: & min'\ (b : as)\ m\ bs \end{cases} \end{aligned} \tag{9.4}$$

函数 min 返回一对值：最小元素和剩余元素列表。这样选择排序就可以实现为：

$$\begin{aligned} sort\ [\,] &= [\,] \\ sort\ xs &= m : (sort\ xs'),\ \text{其中}:\ (m, xs') = min\ xs \end{aligned} \tag{9.5}$$

9.2.1 选择排序的性能

选择排序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，$n + (n-1) + (n-2) + ... + 1$ 次比较，时间复杂度为 $O(\frac{n(n+1)}{2}) = O(n^2)$。和插入排序相比，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 $O(n^2)$。

练习 9.1

1. 下面的尾递归查找最小值实现有何问题？

$$\begin{aligned} min'\ as\ m\ [\,] &= (m, A) \\ min'\ as\ m\ (b : bs) &= \begin{cases} b < m: & min'\ (as + [m])\ b\ bs \\ \text{否则}: & min'\ (as + [b])\ m\ bs \end{cases} \end{aligned}$$

2. 实现非原地和原地的选择排序程序。

9.3 改进

为了支持升序、降序、和不同的比较, 我们可以将比较操作抽出作为参数 $\lhd$。

$$\begin{array}{rcl} sortBy \lhd [\,] & = & [] \\ sortBy \lhd xs & = & m : sortBy \lhd xs', \text{ 其中 : } (m, xs') = minBy \lhd xs \end{array} \tag{9.6}$$

“最小值”也相应地使用传入的 $\lhd$ 进行比较:

$$\begin{array}{rcl} minBy \lhd [x] & = & (x, [\,]) \\ minBy \lhd (x : xs) & = & \begin{cases} x \lhd y : & (x, xs), \text{ 其中 : } (y, ys) = minBy\ xs \\ \text{否则 :} & (y,\ x : ys) \end{cases} \end{array} \tag{9.7}$$

对于一组整数, 传入小于号得到升序结果: $sortBy\ (<)\ [3, 1, 4, ...]$。这里要求比较 $\lhd$ 操作满足严格弱序[52] 条件:

- 非自反性: 对任何 $x, x < x$ 不成立;
- 非对称性: 对任何 x、y, 若 $x < y$, 则 $y < x$ 不成立;
- 传递性: 对任何 x、y、z, 若 $x < y$ 且 $y < z$, 则 $x < z$。

命令式原地选择排序遍历了所有的元素, 可以把最小值查找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]
```

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 $n-1$。另外, 如果第 i 大的元素恰好是 $A[i]$, 我们无需进行交换操作。这样可以进一步改进为:

```
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]
```

9.3.1 鸡尾酒排序

高德纳给出了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。

```
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]
```

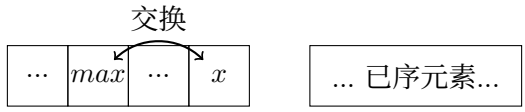


图 9.3: 每次选择最大的元素放到末尾

每次选择最大的元素也可以实现升序排序。进一步，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。这样可以将外重循环次数减半。这一算法称为“鸡尾酒排序”。如图9.4所示，任何时候，左侧和右侧部分都包含了已序元素。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开头和末尾。

```
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] ↔ A[min]
13:        EXCHANGE A[|A| + 1 − i] ↔ A[max]
```

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的元素。也可以用递归的方式实现鸡尾酒排序：

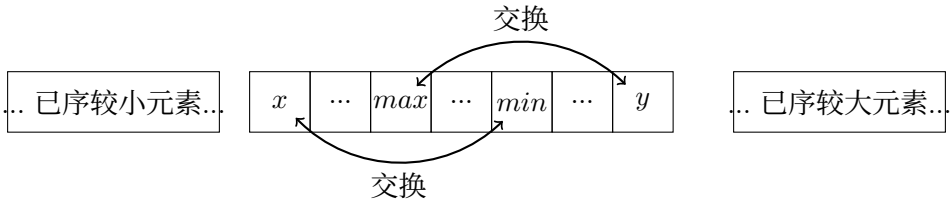


图 9.4: 一次扫描同时定位出最小和最大元素, 然后将它们放到正确的位置

1. 若待排序的序列为空或者仅含有一个元素, 排序结果为原序列;
2. 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。

$$\begin{aligned} sort\ [\,] &= [\,] \\ sort\ [x] &= [x] \\ sort\ xs &= a : (sort\ xs') \mathbin{+\!\!+} [b], \text{其中} : (a, b, xs') = minMax\ xs \end{aligned} \tag{9.8}$$

其中, 函数 $minMax$ 从序列中抽取出最小值和最大值:

$$minMax\ (x : y : xs) = foldr\ sel(min\ x\ y, max\ x\ y, [\,])\ xs \tag{9.9}$$

我们选取头两个元素分别作为已找到的最小值 x_0, 最大值 x_1, 用 $foldr$ 扫描序列, 其中 sel 定义为:

$$sel\ x\ (x_0, x_1, xs) = \begin{cases} x < x_0 : & (x, x_1, x_0 : xs) \\ x_1 < x : & (x_0, x, x_1 : xs) \\ \text{否则} : & (x_0, x_1, x : xs) \end{cases}$$

尽管 $minMax$ 的时间性能是线性时间 $O(n)$ 的, 但 $\mathbin{+\!\!+}[b]$ 的代价较大。如图9.4所示, 令左侧 A 包含较小的已序元素; 右侧 B 包含较大的已序元素。用 A 和 B 作为累积器, 可以将鸡尾酒排序转换为尾递归的:

$$\begin{aligned} sort'\ A\ B\ [\,] &= A \mathbin{+\!\!+} B \\ sort'\ A\ B\ [x] &= A \mathbin{+\!\!+} (x : B) \\ sort'\ A\ B\ (x : xs) &= sort'\ (A \mathbin{+\!\!+} [x_0])\ xs'\ (x_1 : B) \end{aligned} \tag{9.10}$$

其中: $(x_0, x_1, xs') = minMax\ xs$, 我们传入空的 A、B 启动排序: $sort = sort'\ [\,]\ [\,]$。追加操作仅仅发生在 $A \mathbin{+\!\!+} [x_0]$; 而 x_1 则被链结到 B 的前面。每次递归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 x_0 链结到前面而不是追加。我们有如下等价关系:

$$\begin{aligned} A' &= A \mathbin{+\!\!+} [x] \\ &= reverse\ (x : reverse\ A) \\ &= reverse\ (x : \overleftarrow{A}) \\ &= \overleftarrow{x : \overleftarrow{A}} \end{aligned} \tag{9.11}$$

最后执行一次反转操作将 $\overleftarrow{A'}$ 转换回 A'。根据这一思路，可进一步改进如下：

$$\begin{aligned} sort'\ A\ B\ [\,] &= (reverse\ A) +\!\!+ B \\ sort'\ A\ B\ [x] &= (reverse\ x : A) +\!\!+ B \\ sort'\ A\ B\ (x : xs) &= sort'\ (x_0 : A)\ xs'\ (x_1 : B) \end{aligned} \tag{9.12}$$

9.4 继续改进

虽然鸡尾酒排序将循环次数减半，但时间复杂度仍然是 $O(n^2)$ 的。通过比较进行排序，需要检查元素间的大小顺序，外重循环是必须的。为了选出最小元素，必须每次都扫描全部元素么？在查找第一个最小元素时，我们实际遍历了序列，知道哪些元素相对较小，哪些相对较大。但在查找后继的最小元素时，我们没有复用关于相对大小的信息，而是从头开始再次遍历。进一步改进的关键在于重用已有的结果。其中一种方法是来自体育竞赛。

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。我们回到 1978 年，并且想像一种特殊的方法来决定冠军：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德国队进行比赛。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9.5描述了这一情况。

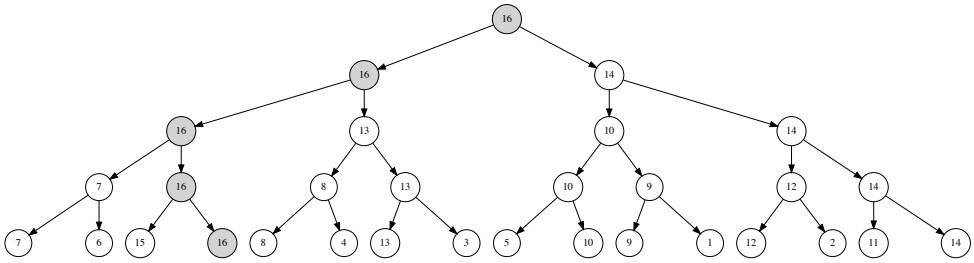


图 9.5: 元素 15 在第一轮就被淘汰

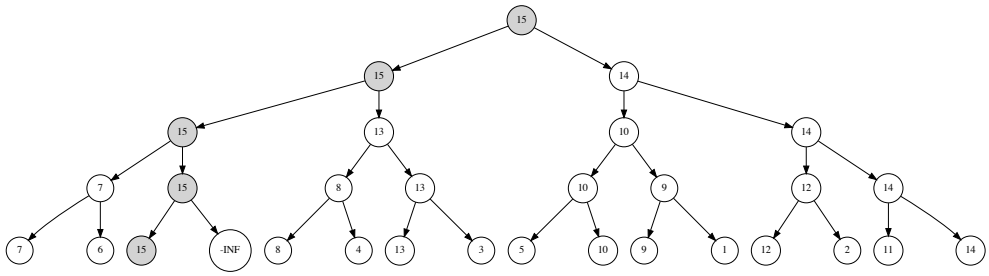
每支队伍有一个代表其实力的数字。数字越大，实力越强。假设数字较大的队永远会战胜数字较小的队（虽然现实中不会这样）。代表冠军的数字为 16，根据假设的规

则, 数字 14 不是亚军, 而是在第一轮就被淘汰的 15。我们需要一种快速的方法在锦标赛树中找到第二大值。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。我们可以把冠军的数字更改成一个很小的值(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, m 是自然数, 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9.5中, 这条路径被标记为灰色, 需要检查的元素包括 $[14, 13, 7, 15]$, 这一思路可以描述如下:

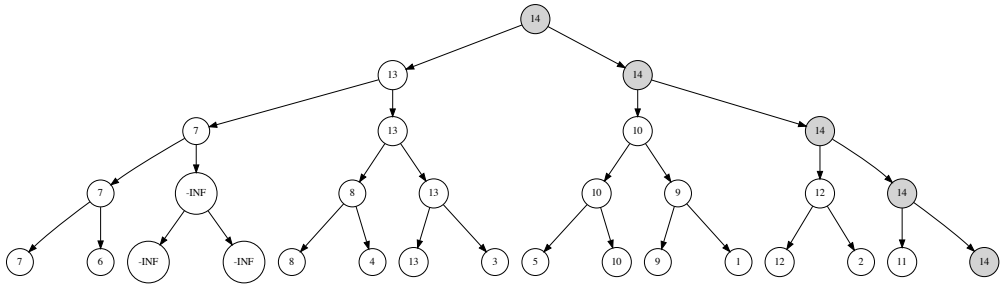
1. 从待排序元素构建一棵锦标赛树, 冠军(最大值)位于树根;
2. 取出树根, 自顶向下沿着冠军路径将最大值替换为 $-\infty$;
3. 自底向上沿着刚才的路径回溯, 找出新的冠军, 并将其置于树根;
4. 重复步骤 2, 直到所有的元素都被取出。

为了对一组元素排序, 我们先从它们构造一棵锦标赛树, 然后不断取出冠军, 图9.6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, 为了方便自底向上回溯, 每个节点需要指向它的父节点。元素数目 n 可能不是恰好 2^m 个。两两比较后, 可能剩余元素没有“对手”, 而“轮空”直接进入下一轮比赛。为了构造锦标赛树, 我们从每个元素构造一棵单一叶子节点的树, 这样就得到了 n 棵二叉树。然后我们每次取出两棵树 t_1、t_2, 构造出一棵更大的二叉树 t。其中 t 的根为 $max(key(t_1), key(t_2))$, 左右子树为 t_1、t_2。重复这一步骤可以得到一组新的树, 每棵新树的高度增加了 1, 如有剩余则进入下一轮。这样一轮过后, 树减半为 $\lfloor \frac{n}{2} \rfloor$。持续同样的操作最终得到一棵锦标赛树。总时间复杂度为 $O(n + \frac{n}{2} + \frac{n}{4} + ...) = O(2n) = O(n)$。

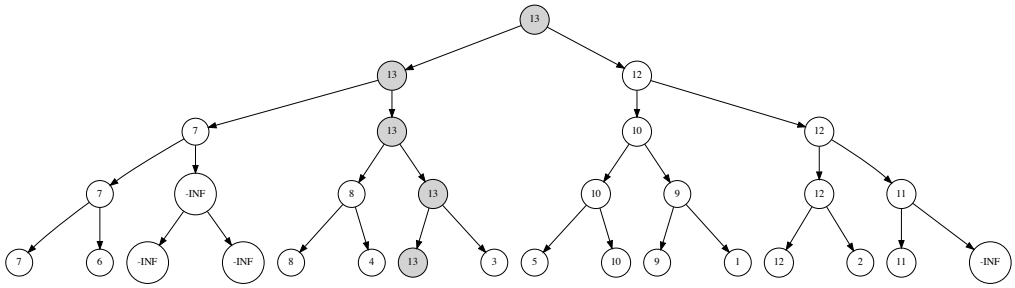
```
1:  function BUILD-TREE(A)
2:      T ← [ ]
3:      for each x ∈ A do
4:          APPEND(T, NODE(NIL, x, NIL))
5:      while |T| > 1 do
6:          T′ ← [ ]
7:          for every t1, t2 ∈ T do
8:              k ← MAX(KEY(t1), KEY(t2))
9:              APPEND(T′, NODE(t1, k, t2))
10:         if |T| is odd then
11:             APPEND(T′, LAST(T))
12:         T ← T′
```



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 9.6: 锦标赛树排序的前几步

```
13:     return T[1]
```

每次取出锦标赛树的根节点后, 我们自顶向下将其替换为 $-\infty$, 然后在通过父节点向上回溯, 找出新的最大值。

```
1: function Pop(T)
2:     m ← Key(T)
3:     Key(T) ← −∞
4:     while T is not leaf do                    ▷ 自顶向下将 m 替换为 −∞
5:         if Key(Left(T)) = m then
6:             T ← Left(T)
7:         else
8:             T ← Right(T)
9:         Key(T) ← −∞
10:    while Parent(T) ≠ NIL do                  ▷ 自底向上决出新冠军
11:        T ← Parent(T)
12:        Key(T) ← Max(Key(Left(T)), Key(Right(T)))
13:    return (m, T)                             ▷ 返回最大元素和更新的树
```

Pop 上下处理两遍, 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 $O(\lg n)$。时间复杂度为 $O(\lg n)$。下面是锦标赛排序的实现。算法首先用 $O(n)$ 时间构建一棵锦标赛树, 然后执行 n 次弹出操作, 逐一从树中取出最大值。每次弹出操作的性能为 $O(\lg n)$, 锦标赛排序的总时间复杂度为 $O(n \lg n)$。

```
1: procedure Sort(A)
2:     T ← Build-Tree(A)
3:     for i ← |A| down to 1 do
4:         (A[i], T) ← Pop(T)
```

也可以用递归实现锦标赛排序。我们复用二叉树的定义。令一棵非空的树为 (l, k, r), 其中 k 为元素, l、r 是左右子树。定义 $wrap\ x = (\varnothing, x, \varnothing)$ 构造一个叶子节点。这样就可以从 n 个元素列表 xs 构造出 n 棵只含一个节点的树: $ts = map\ wrap\ xs$。构造锦标赛树时, 比较两棵树 t_1、t_2, 选择较大的元素作为新的根, 并将 t_1、t_2 作为左右子树。

$$merge\ t_1\ t_2 = (t_1, max\ k_1\ k_2, t_2) \tag{9.13}$$

其中 $k_1 = key\ t_1, k_2 = key\ t_2$, 是两棵树根节点的元素。函数 $build\ ts$ 不断取出两棵树进行合并, 最终构造出锦标赛树。

$$\begin{aligned} build\ [\,] &= \varnothing \\ build\ [t] &= t \\ build\ ts &= build\ (pairs\ ts) \end{aligned} \tag{9.14}$$

其中：

$$\begin{aligned} pairs\ (t_1 : t_2 : ts) &= (merge\ t_1\ t_2) : pairs\ ts \\ pairs\ ts &= ts \end{aligned} \tag{9.15}$$

为了从锦标赛树中取得冠军，我们检查左右子树，看哪一棵子树和根节点的元素相等。然后递归地从子树中取出冠军直到叶子节点。对叶子节点，我们将其中元素替换为 $-\infty$。

$$\begin{aligned} pop\ (\varnothing, k, \varnothing) &= (\varnothing, -\infty, \varnothing) \\ pop\ (l, k, r) &= \begin{cases} k = key\ l: & (l', max\ (key\ l')\ (key\ r), r), \text{其中} l' = pop\ l \\ k = key\ r: & (l, max\ (key\ l)\ (key\ r'), r'), \text{其中} r' = pop\ r \end{cases} \end{aligned} \tag{9.16}$$

排序的过程不断从一棵锦标赛树弹出冠军(降序)：

$$\begin{aligned} sort\ \varnothing &= [\,] \\ sort\ (l, -\infty, r) &= [\,] \\ sort\ t &= (key\ t) : sort\ (pop\ t) \end{aligned} \tag{9.17}$$

练习 9.2

1. 将递归的锦标赛排序实现为升序。
2. 锦标赛树排序可以处理相等元素么？它是稳定排序么？
3. 比较锦标赛树排序和二叉搜索树排序，它们的时间和空间效率如何。
4. 比较堆排序和锦标赛树排序，它们的时间和空间效率如何。

9.4.2 改进为堆排序

锦标赛树淘汰法将基于选择的排序算法时间复杂度提高到 $O(n \lg n)$，达到了基于比较的排序算法上限[51]。这里仍有改进的空间。排序完成后，锦标赛树的所有节点都变成了负无穷，这棵二叉树不再含有任何有用的信息，但却占据了空间。有没有办法在弹出后释放节点呢？如果待排序的元素有 n 个，锦标赛树实际上占用了 $2n$ 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？如果认为根节点的元素为负无穷，则树为空，并将 key 重命名为 top，那么上一节最后给出的式9.17就可以进一步转化为更通用的形式：

$$\begin{aligned} sort\ \varnothing &= [\,] \\ sort\ t &= (top\ t) : sort\ (pop\ t) \end{aligned} \tag{9.18}$$

这和上一章的堆排序定义完全一样。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。使用数组的二叉堆实际上将树结构“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，如左偏堆和伸展堆也只需要 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 附录:例子程序

尾递归实现的选择排序:

```
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
```

鸡尾酒排序:

```
[A] cocktailSort([A] xs) {
    Int n = length(xs)
    for Int i = 0 to n / 2 {
        var (mi, ma) = (i, n - 1 -i)
        if xs[ma] < xs[mi] then swap(xs[mi], xs[ma])
        for Int j = i + 1 to n - 1 - i {
            if xs[j] < xs[mi] then mi = j
            if xs[ma] < xs[j] then ma = j
        }
        swap(xs[i], xs[mi])
        swap(xs[n - 1 - i], xs[ma])
    }
    return xs
}
```

尾递归的鸡尾酒排序:

```
csort xs = cocktail [] [] xs
  where
    cocktail as bs []  = reverse as ++ bs
    cocktail as bs [x] = reverse (x:as) ++ bs
    cocktail as bs xs  = let (mi, ma, xs') = minMax xs
                         in cocktail (mi:as) (ma:bs) xs'

minMax (x:y:xs) = foldr sel (min x y, max x y, []) xs
  where
    sel x (mi, ma, ys) | x < mi = (x, ma, mi:ys)
                       | ma < x = (mi, x, ma:ys)
                       | otherwise = (mi, ma, x:ys)
```

复用二叉树构造锦标赛树:

```
Node<T> build([T] xs) {
    [T] ts = []
    for x in xs {
        append(ts, Node(null, x, null))
    }
```

```
    while length(ts) >  1 {
        [T] ts' = []
        for l, r in ts {
            append(ts', Node(l, max(l.key, r.key), r))
        }
        if odd(length(ts)) then append(ts', last(ts))
        ts = ts'
    }
    return ts[0];
}
```

从锦标赛树取出冠军:

```
T pop(Node<T> t) {
    T m = t.key
    t.key = -INF
    while not isLeaf(t) {
        t = if t.left.key == m then t→left else t→right
        t.key = -INF
    }
    while (t.parent ≠ null) {
        t = t.parent
        t.key = max(t.left.key, t.right.key)
    }
    return (m, t);
}
```

锦标赛树排序:

```
void sort([A] xs) {
    Node<T> t = build(xs)
    for Int n = length(xs) - 1 downto 0 {
        (xs[n], t) = pop(t)
    }
}
```

递归的锦标赛排序(降序):

```
data Tr a = Empty | Br (Tr a) a (Tr a)

data Infinite a = NegInf | Only a | Inf deriving (Eq, Ord)

key (Br _ k _ ) = k

wrap x = Br Empty (Only x) Empty

merge t1@(Br _ k1 _) t2@(Br _ k2 _) = Br t1 (max k1 k2) t2

fromList = build ∘ (map wrap) where
  build [] = Empty
  build [t] = t
  build ts = build (pairs ts)
  pairs (t1:t2:ts) = (merge t1 t2) : pair ts
  pairs ts = ts
```

```
pop (Br Empty _ Empty) = Br Empty NegInf Empty
pop (Br l k r) | k == key l = let l' = pop l in Br l' (max (key l') (key r)) r
               | k == key r = let r' = pop r in Br l (max (key l) (key r')) r'

toList Empty = []
toList (Br _ Inf _) = []
toList t@(Br _ Only k _) = k : toList (pop t)

sort = toList ∘ fromList
```

第十章　二项式堆, 斐波那契堆、配对堆

10.1　简介

二叉堆使用二叉树存储元素, 将二叉树扩展成 k 叉树[54]$(k > 2)$ 甚至多棵树可得到更丰富的堆结构。本章介绍二项式堆, 它由多棵 k 叉树的森林组成。如果延迟执行二项式堆的某些操作, 就可以得到斐波那契堆。斐波那契堆将堆合并的性能从对数时间复杂度提升到常数时间, 这对于图算法很重要。本章还介绍配对堆。它在实际中拥有最好的性能。

10.2　二项式堆

二项式堆得名于牛顿二项式。它由一组 k 叉树的森林组成, 每棵树的大小为二项式展开中的各项系数。牛顿证明了形如 $(a+b)^n$ 的二项式展开后, 各项系数可以表示为:

$$(a+b)^n = a^n + \binom{n}{1}a^{n-1}b + ... + \binom{n}{n-1}ab^{n-1} + b \tag{10.1}$$

当 n 为自然数时, 各项系数就呈现出帕斯卡三角形中的一行, 如下图所示[1] [55]。

```
     1
    1 1
   1 2 1
  1 3 3 1
1 4 6 4 1
...
```

有多种方法可以产生一系列二项式系数, 其中一种是使用递归。帕斯卡三角形中第一行为 1, 任何一行的两端为 1, 其它数字是上一行中左上和右上数字之和。

[1] 中国称“贾宪”三角形。贾宪(1010-1070), 牛顿在 1665 年证明了 n 为有理数时的情形, 欧拉后来将 n 推广到实数。

10.2.1 二项式树

一棵二项式树是一棵多叉树, 并带有一个整数的秩(rank)。记秩为 0 的二项式树为 B_0, 秩为 n 的二项式树为 B_n。

1. B_0 树只包含一个节点;
2. B_n 树由两棵 B_{n-1} 树组成, 其中根节点元素较大的一棵是另一棵最左侧的子树。如图10.1所示。

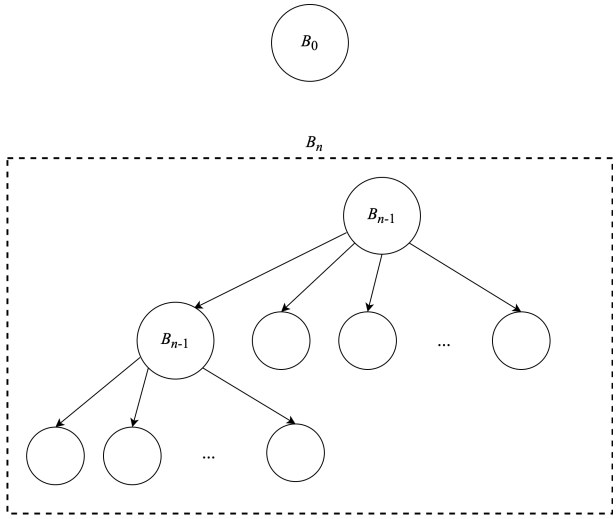


图 10.1: 二项式树

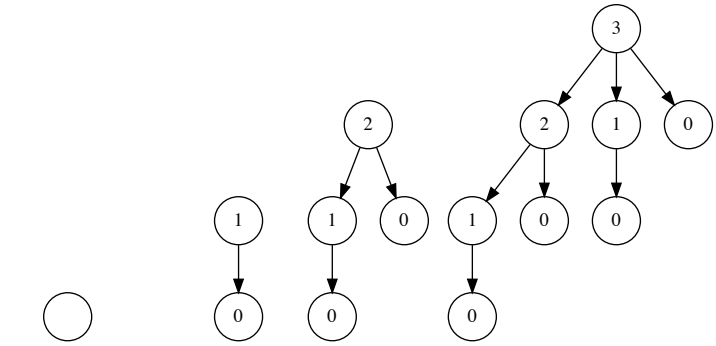
图10.2给出了秩为 0 到 4 的二项式树例子:

观察这些二项式树, 可以发现 B_n 中每行的节点数目恰好是二项式系数。例如 B_4 第 0 层有 1 个节点(根节点), 第 1 层有 4 个节点, 第 2 层有 6 个节点, 第 3 层有 4 个节点, 第 4 层有 1 个节点。它们恰好是帕斯卡三角形的第 4 行(从第 0 行开始):1、4、6、4、1。这就是二项式树名字的由来。进一步我们可以得知二项式树 B_n 中含有 2^n 个元素。

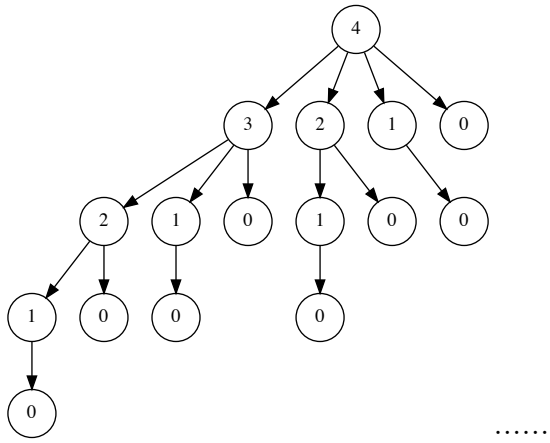
一个二项式堆包含一组二项式树(二项式树森林), 它满足如下性质:

1. 每棵树都满足**堆性质**, 对于小顶堆, 任意节点元素都不小于($\geq$)父节点元素;
2. 堆中任何两棵二项式树的秩都不同。

从性质 2 可以导出一个结果:含有 n 个元素的二项式堆, 如果将 n 转换为二进制数 $(a_m...a_1a_0)_2$, 其中 a_0 是最低位(LSB), a_m 是最高位(MSB), 若 $a_i = 0$, 则堆中不存在秩为 i 的二项式树, 若 $a_i = 1$, 则堆中一定含有一棵秩为 i 的树。例如, 设二项式堆含有 5 个元素, 5 的二进制为 101, 堆中含有两棵二项式树, 一棵秩为 0、一棵秩为 2。



(a) B_0 (b) B_1 (c) B_2 (d) B_3



(e) B_4

图 10.2: 秩为 0、1、2、3、4……的二项式树

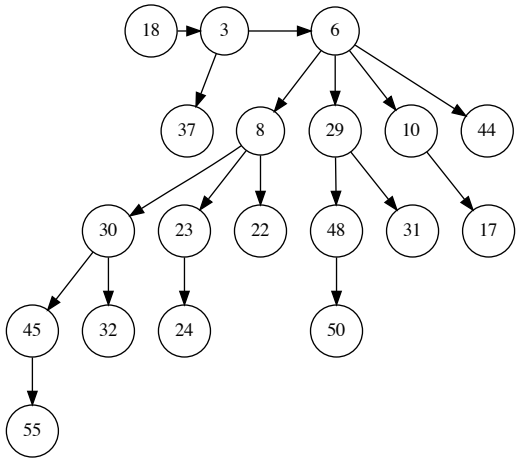


图 10.3: 含有 19 个元素的二项式堆

图10.3中的二项式堆含有 19 个元素, $19 = (10011)_2$, 含有一棵 B_0 树、一棵 B_1 树、一棵 B_4 树。

我们将二项式树定义为多叉树, 带有一个根节点元素 k、秩 r、和若干子树 ts, 记为 (r, k, ts)。定义二项式堆为按照秩递增的二项式树的列表:

```
data BiTree a = Node Int a [BiTree a]

type BiHeap a = [BiTree a]
```

有一种叫做“左侧孩子, 右侧兄弟”[4] 的方法, 可以复用二叉树的结构来定义多叉树。每个节点包含左侧和右侧部分:左侧部分指向节点的第一棵子树, 右侧部分指向兄弟节点。所有兄弟节点组成一个链表, 如图10.4所示。也可以直接利用数组或列表来表示一个节点的子树。

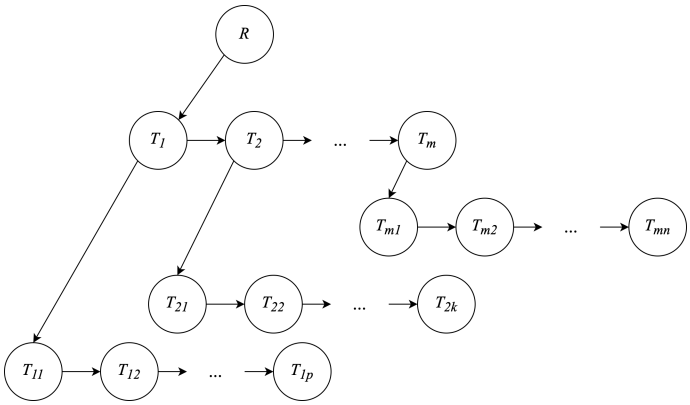


图 10.4: R 为根节点,$T_1, T_2, ..., T_m$ 为 R 的子树。R 的左侧为 T_1,右侧为空。$T_{11}, ..., T_{1p}$ 为 T_1 的子树。T_1 的左侧是子树 T_{11}, 右侧是兄弟节点 T_2。T_2 的左侧是子树 T_{21}, 右侧是兄弟节点。

10.2.2 树的链接

我们定义链接操作从两棵二项式树 B_n 构造出 B_{n+1}。比较两棵树的根节点元素, 选择较小的作为新的根, 然后将另一棵置于其它子树前面, 如图10.5所示。

$$link\ (r, x, ts)\ (r, y, ts') = \begin{cases} x < y: & (r+1, x, (r, t, ts') : ts) \\ \text{否则}: & (r+1, y, (r, x, ts) : ts') \end{cases} \tag{10.2}$$

使用“左侧孩子, 右侧兄弟”的实现如下。链接操作可以在常数时间内完成。

```
1: function LINK(x, y)
2:     if KEY(y) < KEY(x) then
3:         Exchange x ↔ y
4:     SIBLING(y) ← SUB-TREES(T1)
```

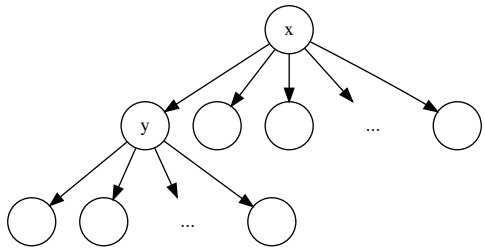


图 10.5: 如果 $x < y$, 将 y 作为 x 的第一棵子树。

5: SUB-TREES$(x) \leftarrow y$
6: PARENT$(y) \leftarrow x$
7: RANK$(x) \leftarrow$ RANK$(y) + 1$
8: **return** x

练习 10.1

1. 编程产生帕斯卡三角形
2. 证明二项式树 B_n 中第 i 行的节点数为 $\binom{n}{i}$。
3. 证明二项式树 B_n 中含有 2^n 个节点。
4. 用容器保存子树, 实现二项式树的链接。这种方式有何问题, 怎样解决?

10.2.3 插入

我们令堆中二项式树按照秩递增排列, 并在插入新树时保持秩的顺序:

$$\begin{aligned} ins\ t\ [\,] &= [t] \\ ins\ t\ (t' : ts) &= \begin{cases} rank\ t < rank\ t' : & t : t' : ts \\ rank\ t' < rank\ t : & t' : ins\ t\ ts \\ \text{否则} : & ins\ (link\ t\ t')\ ts \end{cases} \end{aligned} \tag{10.3}$$

其中 $rank\ (r, k, ts) = r$, 获取一棵二项式树的秩。如果堆为空 $[\,]$, 则新树 t 成为堆中唯一的树; 否则, 我们比较 t 和堆中第一棵树 t' 的秩。如果 t 的秩较小, 则 t 成为第一棵树; 如果 t' 的秩较小, 我们递归地将 t 插入到剩余的树中; 如果秩相等, 就将 t、t' 链接成一棵更大的树, 然后递归地插入到剩余的树中。如果有 n 个元素, 堆中最多有 $O(\lg n)$ 棵二项式树。ins 最多执行 $O(\lg n)$ 次常数时间的链接。其时间复杂度为 $O(\lg n)$[2]。使用 ins, 我们可以定义二项式堆的插入算法。先将待插入元素 x 放入一棵只有一个叶子节点树中, 然后再插入到堆中:

$$insert\ x = ins\ (0, x, [\,]) \tag{10.4}$$

[2]这一过程和两个二进制数的加法相似, 可以引出一类问题: 数值表示(numeric representation)[3]。

这一定义是克里化的。我们可以利用叠加操作将若干元素插入到堆中:

$$fromList = foldr\ insert\ [\,] \tag{10.5}$$

对应的"左侧孩子, 右侧兄弟"实现如下:

```
1: function INSERT-TREE(T, H)
2:     ⊥← p ← NODE(0, NIL, NIL)
3:     while H ≠ NIL 且 RANK(H) ≤ RANK(T) do
4:         T_1 ← H
5:         H ← SIBLING(H)
6:         if RANK(T) = RANK(T_1) then
7:             T ← LINK(T, T_1)
8:         else
9:             SIBLING(p) ← T_1
10:            p ← T_1
11:    SIBLING(p) ← T
12:    SIBLING(T) ← H
13:    return REMOVE-FIRST(⊥)

14: function REMOVE-FIRST(H)
15:    n ← SIBLING(H)
16:    SIBLING(H) ← NIL
17:    return n
```

10.2.4 堆合并

合并两个二项式堆相当于合并两个二项式树森林。合并结果中没有秩相同的树, 并且按照秩递增。合并过程和归并排序类似。每次从两个堆中各取出第一棵树, 比较它们的秩, 将较小的一棵放入结果中。如果两棵树的秩相等, 我们将它们链接成为一棵较大的树, 然后递归插入到合并结果中。

$$\begin{aligned}
merge\ ts_1\ [\,] &= ts_1 \\
merge\ [\,]\ ts_2 &= ts_2 \\
merge\ (t_1 : ts_1)\ (t_2 : ts_2) &= \begin{cases}
rank\ t_1 < rank\ t_2 : & t_1 : (merge\ ts_1\ (t_2 : ts_2)) \\
rank\ t_2 < rank\ t_1 : & t_2 : (merge\ (t_1 : ts_1)\ ts_2) \\
\text{否则} : & ins\ (link\ t_1\ t_2)\ (merge\ ts_1\ ts_2)
\end{cases}
\end{aligned} \tag{10.6}$$

当 t_1、t_2 秩相同时, 我们也可以将链接后的树插入回任意一个堆, 然后递归合并:

$$merge\ (ins\ (link\ t_1\ t_2)\ ts_1)\ ts_2$$

用这种方式可以消除递归,用迭代的方式实现堆合并:

```
1: function MERGE(H_1, H_2)
2:     H ← p ← NODE(0, NIL, NIL)
3:     while H_1 ≠ NIL 且 H_2 ≠ NIL do
4:         if RANK(H_1) < RANK(H_2) then
5:             SIBLING(p) ← H_1
6:             p ← SIBLING(p)
7:             H_1 ← SIBLING(H_1)
8:         else if RANK(H_2) < RANK(H_1) then
9:             SIBLING(p) ← H_2
10:            p ← SIBLING(p)
11:            H_2 ← SIBLING(H_2)
12:        else                                          ▷ 秩相等
13:            T_1 ← H_1, T_2 ← H_2
14:            H_1 ← SIBLING(H_1), H_2 ← SIBLING(H_2)
15:            H_1 ← INSERT-TREE(LINK(T_1, T_2), H_1)
16:    if H_1 ≠ NIL then
17:        SIBLING(p) ← H_1
18:    if H_2 ≠ NIL then
19:        SIBLING(p) ← H_2
20:    return REMOVE-FIRST(H)
```

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。如果没有秩相同的树,则合并时间为 $O(m_1 + m_2)$。如果存在秩相同的树,最多需要调用 $O(m_1 + m_2)$ 次 ins。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 是两个堆各自的元素数,且 $\lfloor \lg n_1 \rfloor + \lfloor \lg n_2 \rfloor \leq 2\lfloor \lg n \rfloor$, 其中 $n = n_1 + n_2$。最终合并的复杂度为 $O(\lg n)$。

10.2.5 弹出

二项式堆中,每棵树的根节点保存了树中的最小元素。但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,需要在全部根节点中查找。因为堆中有 $O(\lg n)$ 棵树,所以获取最小值的时间复杂度为 $O(\lg n)$。但是弹出操作不仅找出最小元素,还需要将其删除并保持堆性质。设堆中的二项式树为 $B_i, B_j, ..., B_p, ..., B_m$。设 B_p 的根节点为堆中最小元素。将其删除后会产生 p 棵子二项式树,秩为 $p-1, p-2, ..., 0$。我们可以将 p 棵子树逆序,形成一个新二项式堆 H_p。除去 B_p 的树也构成一个二项式堆 $H' = H - [B_p]$。将 H_p 和 H' 合并就可以得到最终结果,如图10.6所示。我们首先定义从堆中寻找最小元素的操作:

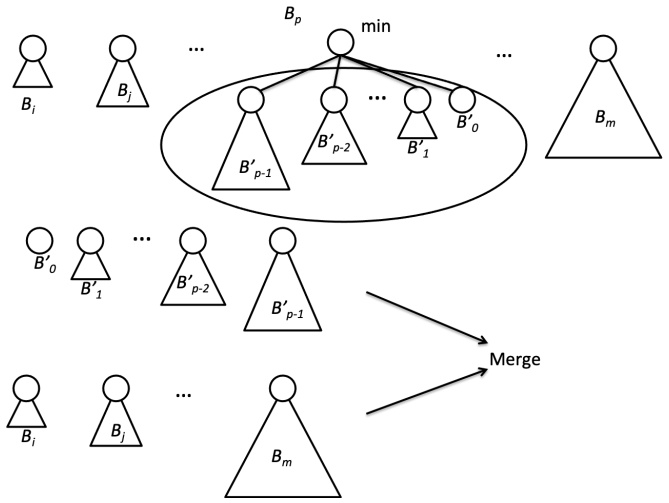


图 10.6: 二项式堆的弹出操作

$$top\ (t : ts) = foldr\ f\ (key\ t)\ ts \tag{10.7}$$

其中

$$f\ (r, x, ts)\ y = min\ x\ y$$

这相当于遍历堆中的所有树, 找出根节中存储的最小值:

```
1: function TOP(H)
2:     m ← ∞
3:     while H ≠ NIL do
4:         m ← MIN(m, KEY(H))
5:         H ← SIBLING(H)
6:     return m
```

为了实现弹出, 我们需要从堆中分离出最小元素所在的树:

$$\begin{aligned} min'\ [t] &= (t, [\,]) \\ min'\ (t : ts) &= \begin{cases} key\ t < key\ t' : & (t, ts), \text{其中} : (t', ts') = min'\ ts \\ \text{否则} : & (t', t : ts') \end{cases} \end{aligned} \tag{10.8}$$

其中 $key\ (r, k, ts) = k$ 获取二项式树中的根节点元素。min' 的结果为一对值: 最小元素所在的树, 和其它的树。接下来就可以定义弹出操作:

$$pop\ H = (k, merge\ (reverse\ ts)\ H'), \text{其中} : ((r, k, ts), H') = min'\ H \tag{10.9}$$

对应的迭代实现为:

```
1: function POP(H)
```

```
    (T_m, H) ← EXTRACT-MIN(H)
    H ← MERGE(H, REVERSE(SUB-TREES(T_m)))
    SUB-TREES(T_m)
    return (KEY(T_m), H)
```

其中反转操作的实现见第一章, EXTRACT-MIN 的迭代实现如下:

```
function EXTRACT-MIN(H)
    H' ← H, p ← NIL
    T_m ← T_p ← NIL
    while H ≠ NIL do
        if T_m = NIL 或 KEY(H) < KEY(T_m) then
            T_m ← H
            T_p ← p
        p ← H
        H ← SIBLING(H)
    if T_p ≠ NIL then
        SIBLING(T_p) ← SIBLING(T_m)
    else
        H' ← SIBLING(T_m)
    SIBLING(T_m) ← NIL
    return (T_m, 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。

$$sort = heapSort \circ fromList \tag{10.10}$$

其中 $heapSort$ 实现如下:

$$\begin{aligned} heapSort\ [\,] &= [\,] \\ heapSort\ H &= k : (heapSort\ H'), \text{其中} : (k, H') = pop\ H \end{aligned} \tag{10.11}$$

二项式堆的插入、合并的时间复杂度在最坏情况下是 $O(\lg n)$。他们的分摊复杂度为常数时间, 我们这里略去了分摊复杂度的证明。

10.3 斐波那契堆

二项式堆的名字来自二项式展开, 斐波那契堆的名字来自斐波那契数列[3]。斐波那契堆本质上是一个惰性二项式堆。但这并不意味着二项式堆在支持惰性求值的环境下

[3]Michael L. Fredman 和 Robert E. Tarjan 在证明这种堆的时间复杂度时使用了斐波那契数列的性质, 他们于是给这种堆命名为“斐波那契堆”[4]。

自动就成为了斐波那契堆。惰性环境仅提供了实现上的便利[56]。除弹出操作外, 斐波那契堆所有操作的分摊复杂度都可以达到常数时间[57]。

操作	二项式堆	斐波那契堆
插入	$O(\lg n)$	$O(1)$
合并	$O(\lg n)$	$O(1)$
获取堆顶	$O(\lg n)$	$O(1)$
弹出	$O(\lg n)$	分摊 $O(\lg n)$

表 10.1: 斐波那契堆和二项式堆的(分摊)复杂度对比

向二项式堆插入新元素 x 时, 我们将 x 放入只有一个叶子节点的树中, 然后插入到森林中。树按照秩递增的顺序插入, 如果秩相等, 则进行链接, 然后递归插入。时间复杂度为 $O(\lg n)$。使用惰性策略, 我们将按秩有序插入和链接等操作推迟进行。将 x 所在的树直接加入森林中。为了快速获得堆顶元素, 我们需要记录哪一棵树的根节点保存了最小元素。一个斐波那契堆或者为空 $\varnothing$, 或者是若干树的森林, 记为 (n, t_m, ts)。其中含有最小元素的树被单独记录为 t_m, 堆中元素个数记录为 n, 其余二项式树的列表为 ts。下面的例子程序定义了斐波那契堆(复用了二项式树的定义):

```
data FibHeap a = E | FH { size :: Int
                        , minTree :: BiTree a
                        , trees :: [BiTree a]}
```

这样就可以用常数时间获取堆顶元素:$top\ H = key\ minTree\ H$。

10.3.1 插入

我们将插入定义为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:

$$insert\ x\ H = merge\ (singleton\ x)\ H$$

或写成克里化的形式:

$$insert = merge \circ singleton \tag{10.12}$$

其中 $singleton$ 从 x 构建仅含有一个元素的树:

$$singleton\ x = (1, (1, x, [\,]), [\,])$$

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(k, H)
2: $x \leftarrow$ SINGLETON(k) ▷ 将 k 装入一棵树
3: ADD(x, TREES(H))
4: $T_m \leftarrow$ MIN-TREE(H)

5: **if** $T_m = \text{NIL}$ 或 $k < \textsc{Key}(T_m)$ **then**
6: $\textsc{Min-Tree}(H) \leftarrow x$
7: $\textsc{Size}(H) \leftarrow \textsc{Size}(H) + 1$

其中 $\textsc{Trees}(H)$ 获取堆 H 中所有树的列表, $\textsc{Min-Tree}(H)$ 记录了 H 中最小元素所在的树。

10.3.2 合并

和二项式堆不同, 我们在合并时将链接操作推迟到将来, 仅仅将两个堆中的树放到一起, 然后比较出新的含有最小元素的树。

$$\begin{aligned} merge\ h\ \varnothing &= h \\ merge\ \varnothing\ h &= h \\ merge\ (n, t_m, ts)\ (n', t'_m, ts') &= \begin{cases} key\ t_m < key\ t'_m: & (n+n', t_m, t'_m : ts +\!\!+ ts') \\ \text{否则}: & (n+n', t'_m, t_m : ts +\!\!+ ts') \end{cases} \end{aligned} \tag{10.13}$$

当两个堆都不为空时,这一实现中的 $+\!\!+$ 操作和其中一个堆中树的棵数成正比。如果使用双向链表, 则可以把堆合并操作提高到常数时间。下面的例子程序使用双向链表定义了斐波那契堆:

```
data Node<K> {
    K key
    Int rank
    Node<k> next, prev, parent, subTrees
}

data FibHeap<K> {
    Int size
    Node<K> minTree, trees
}
```

这样就可以实现常数时间的合并操作:

1: **function** $\textsc{Merge}(H_1, H_2)$
2: $H \leftarrow \textsc{Fib-Heap}$
3: $\textsc{Trees}(H) \leftarrow \textsc{Concat}(\textsc{Trees}(H_1), \textsc{Trees}(H_2))$
4: **if** $\textsc{Key}(\textsc{Min-Tree}(H_1)) < \textsc{Key}(\textsc{Min-Tree}(H_2))$ **then**
5: $\textsc{Min-Tree}(H) \leftarrow \textsc{Min-Tree}(H_1)$
6: **else**
7: $\textsc{Min-Tree}(H) \leftarrow \textsc{Min-Tree}(H_2)$
$\textsc{Size}(H) = \textsc{Size}(H_1) + \textsc{Size}(H_2)$
8: **return** H

9: **function** $\textsc{Concat}(s_1, s_2)$

10: $e_1 \leftarrow \textsc{Prev}(s_1)$
11: $e_2 \leftarrow \textsc{Prev}(s_2)$
12: $\textsc{Next}(e_1) \leftarrow s_2$
13: $\textsc{Prev}(s_2) \leftarrow e_1$
14: $\textsc{Next}(e_2) \leftarrow s_1$
15: $\textsc{Prev}(s_1) \leftarrow e_2$
16: **return** s_1

10.3.3 弹出

我们在合并时推迟了树的链接，接下来需要在弹出时将其“补偿”回来。我们定义这一过程为树的归并。首先考虑这样一个问题：给定若干 2 的整数次幂，如：$L = [2, 1, 1, 4, 8, 1, 1, 2, 4]$，我们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 $[8, 16]$。表10.2给出了归并的步骤。第一列表示每次“扫描”到的数字；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。这个数的归并的过程可以用叠加来实现：

数字	比较、相加	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

$$consolidate = foldr\ melt\ [\,] \tag{10.14}$$

其中 $melt$ 定义为：

$$\begin{array}{rcl} melt\ x\ [\,] & = & x \\ melt\ x\ (x' : xs) & = & \begin{cases} x = x' : & melt\ 2x\ xs \\ x < x' : & x : x' : xs \\ x > x' : & x' : melt\ x\ xs \end{cases} \end{array} \tag{10.15}$$

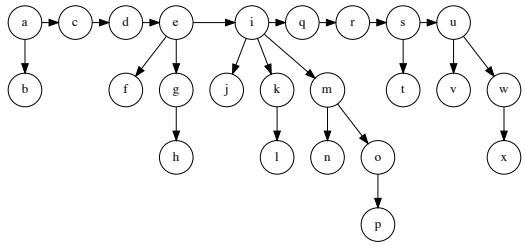
令 $n = sum\ L$，为所有数字的和，$consolidate$ 相当于把 n 表示为二进制数，如果第 i 位上的数字是 1，则最终列表中包含 2^i 这个数（i 从 0 开始）。例如 $sum[2,1,1,4,8,1,1,2,4] = 24$，写成二进制是 11000，第 3 和第 4 位上是 1，所以最终列表中包含 $2^3 = 8, 2^4 = 16$。用类似的方法可以实现树的归并。我们需要比较秩，并把秩相同的树链接起来：

$$\begin{aligned} melt\ t\ [\,] &= [t] \\ melt\ t\ (t' : ts) &= \begin{cases} rank\ t = rank\ t' : & melt\ (link\ t\ t')\ ts \\ rank\ t < rank\ t' : & t : t' : ts \\ rank\ t > rank\ t' : & t' : melt\ t\ ts \end{cases} \end{aligned} \tag{10.16}$$

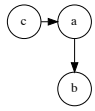
图10.7给出了斐波那契堆树归并过程的步骤，和表10.2对比可以看出它们是相似的。我们也可以使用一个辅助数组 A 来进行归并。$A[i]$ 保存秩为 i 的树。在遍历堆中的树时，如果遇到另一棵秩为 i 的树，我们就将它们链接起来得到一棵秩为 $i+1$ 的树。然后将 $A[i]$ 清空，并接着检查 $A[i+1]$ 是否为空，若不为空，就再次进行链接。遍历完成后，A 中就保存了归并后的结果。

```
1:  function CONSOLIDATE(H)
2:      R ← MAX-RANK(SIZE(H))
3:      A ← [NIL, NIL, ..., NIL]                          ▷ 共 R 个
4:      for each T in TREES(H) do
5:          r ← RANK(T)
6:          while A[r] ≠ NIL do
7:              T' ← A[r]
8:              T ← LINK(T, T')
9:              A[r] ← NIL
10:             r ← r + 1
11:         A[r] ← T
12:     T_m ← NIL
13:     TREES(H) ← NIL
14:     for each T in A do
15:         if T ≠ NIL then
16:             append T to TREES(H)
17:             if T_m = NIL or KEY(T) < KEY(T_m) then
18:                 T_m ← T
19:     MIN-TREE(H) ← T_m
```

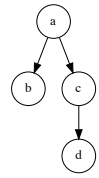
将堆中所有二项式树归并后，斐波那契堆就变成了二项式堆。此时堆中的树合并为 $O(\lg n)$ 棵。MAX-RANK(n) 返回 n 个元素的堆中最大可能的秩 R。根据二项式堆的结论，秩最大的树 B_R 有 2^R 个元素。我们有：$2^R \le n < 2^{R+1}$。我们推测它一个大致



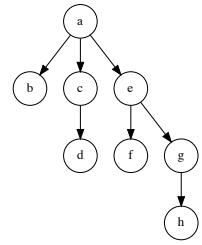
归并前



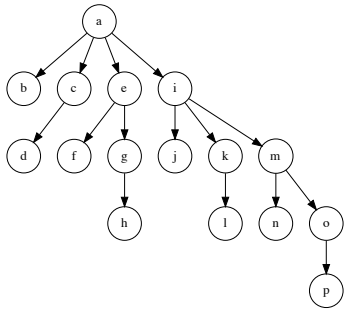
第 1、2 步



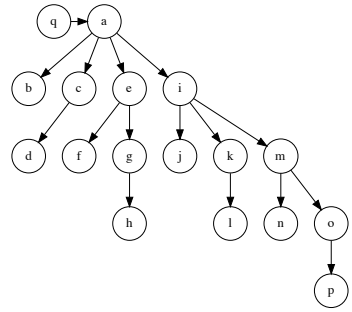
第 3 步, d 先链接到 c, 然后链接到 a。



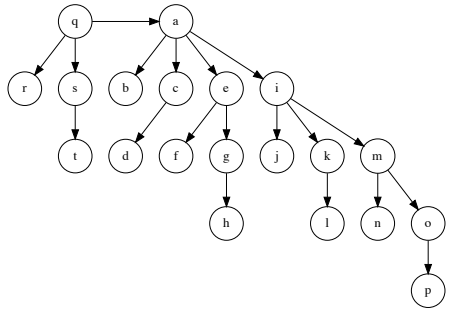
第 4 步



第 5 步



第 6 步



第 7、8 步, r 先链接到 q, 然后 s 链接到 q。

图 10.7: 树归并的步骤

上限为 $R \leq \log_2 n$。我们稍后给出 R 的更准确的估计。我们需要额外扫描一遍所有树的根节点，找到存有最小元素的树。我们可以复用 (10.8) 定义的 min' 分离出堆顶元素所在的树。

$$\begin{array}{rcl} pop\ (1, (0, x, [\,]), [\,]) & = & (x, [\,]) \\ pop\ (n, (r, x, ts_m), ts) & = & (x, (n-1, t_m, ts')) \end{array} \tag{10.17}$$

其中 $(t_m, ts') = min'\ consolidate\ (ts_m +\!\!+ ts)$。注意到 $+\!\!+$ 的时间复杂度是 $O(|ts_m|)$，和最小值所在的树中的子树棵数成正比。对应的命令式实现如下：

```
1: function POP(H)
2:     T_m ← MIN-TREE(H)
3:     for each T in SUB-TREES(T_m) do
4:         append T to TREES(H)
5:         PARENT(T) ← NIL
6:     remove T_m from TREES(H)
7:     SIZE(H) ← SIZE(H) - 1
8:     CONSOLIDATE(H)
9:     return (KEY(T_m), H)
```

我们使用“势能方法”分析弹出算法的分摊性能。回忆物理学中重力势能的定义：

$$E = mgh \tag{10.18}$$

如图10.8所示，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：

$$W = mg(h' - h) + W_f$$

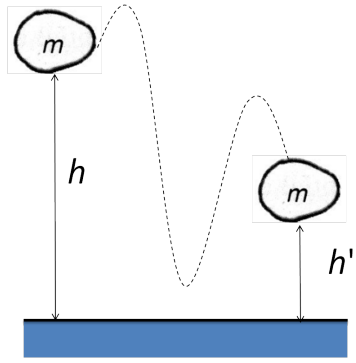


图 10.8: 重力势能

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能为 $\Phi(H)$。这个势能是由迄今为止的插入、合并操作累积的。经过树的归并，形成

了新的堆 H', 由此计算新的势能 $\Phi(H')$。$\Phi(H')$ 和 $\Phi(H)$ 的差再加上树归并消耗的部分就可以给出弹出的分摊复杂度。定义势能为:

$$\Phi(H) = t(H) \tag{10.19}$$

其中 $t(H)$ 是堆中树的棵数。对于 n 个节点的斐波那契堆, 令堆中所有树秩的上限为 $R(n)$。归并后, 堆中树的棵数最多为 $t(H') = R(n) + 1$。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 $R(n) + t(H) - 1$ 棵树。设弹出操作的时间复杂度为 T, 树归并的时间复杂度为 T_c, 我们推导分摊性能如下:

$$\begin{aligned} T &= T_c + \Phi(H') - \Phi(H) \\ &= O(R(n) + t(H) - 1) + (R(n) + 1) - t(H) \\ &= O(R(n)) \end{aligned} \tag{10.20}$$

插入、合并、弹出操作可以确保斐波那契堆中的所有树都为二项式树, 此时 $R(n)$ 的上限为 $O(\lg n)$。

10.3.4 提升优先级

提升优先级是堆的一个实际应用。用堆管理若干任务, 某个任务需要提前执行, 为此我们希望将代表其优先级的数值减小, 使得任务更接近堆顶。某些图算法, 例如最小生成树算法和 Dijkstra 算法都依赖这一操作[4]。并且我们需要其分摊性能达到常数时间。令 x 指向堆 H 中的某个节点, 我们希望将它的值减小为 k。如图10.9, 若节点 x 的值小于父节点 y, 将子树 x 切下并添加到堆(森林)中。尽管这样可以使得每棵树的父节点仍然存有最小值, 但切除节点后的树不再是二项式树。如果失去了很多子树, 就无法保证合并操作的性能。为了解决这一问题, 我们给斐波那契堆增加一个限制条件:

当一个节点第二次失去了某个子节点时, 立即将它切下并添加到堆(森林)中。

```
1: function DECREASE(H, x, k)
2:     KEY(x) ← k
3:     p ← PARENT(x)
4:     if p ≠ NIL and k < KEY(p) then
5:         CUT(H, x)
6:         CASCADE-CUT(H, p)                    ▷ 回溯切除
7:     if k < TOP(H) then
8:         MIN-TREE(H) ← x
```

CASCADE-CUT 使用一个标记来记录它此前是否失去过子节点。并在 CUT 中清除这一标记。

```
1: function CUT(H, x)
```

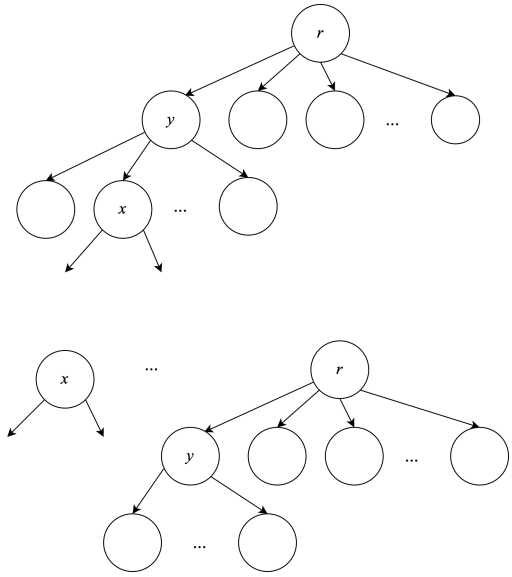


图 10.9: 若 $key\ x < key\ y$, 将 x 切下, 然后添加到堆中。

```
2:    p ← Parent(x)
3:    remove x from p
4:    Rank(p) ← Rank(p) - 1
5:    add x to Trees(H)
6:    Parent(x) ← NIL
7:    Mark(x) ← False
```

回溯切除时, 若节点 x 被标记了, 说明它此前已经失去了子节点。我们需要继续回溯切除, 直到根节点。

```
1: function Cascade-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:            Cascade-Cut(H, p)
```

练习 10.2

证明 Decrease 的分摊复杂度为常数时间 $O(1)$。

10.3.5 斐波那契堆的命名

我们尚未给出 MAX-RANK(n) 的实现。它定义了 n 个元素的斐波那契堆中所有树的秩的上限。

引理 10.3.1. 对于斐波那契堆中的任何树 x, 若其秩为 k, (即: $k = rank(x)$), $|x|$ 为树中元素个数, 则:

$$|x| \geq F_{k+2} \tag{10.21}$$

其中 F_k 为斐波那契数列中的第 k 项:

$$\begin{aligned} F_0 &= 0 \\ F_1 &= 1 \\ F_k &= F_{k-1} + F_{k-2} \end{aligned}$$

证明. 考虑节点 x 的全部 k 棵子树: $y_1, y_2, ..., y_k$。顺序是被链接到 x 的时间先后。其中 y_1 最早加入, y_k 最新加入。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。因为我们只把秩相同的树链接起来, 所以在这一时刻, 有:

$$rank(y_i) = rank(x) = i - 1$$

此后 y_i 最多只能失去一个子节点(通过 DECREASE), 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:

$$rank(y_i) \geq i - 2$$

令 s_k 为 x 的子节点个数**可能的最小值**, 其中 $k = rank(x)$。对于边界情况, 有 $s_0 = 1$, $s_1 = 2$。也就是说: 秩为 0 的树, 最少有 1 个节点, 秩为 1 的树最少有 2 个节点, 秩为 k 的树, 最少有 s_k 个节点:

$$\begin{aligned} |x| &\geq s_k \\ &= 2 + s_{rank(y_2)} + s_{rank(y_3)} + ... + s_{rank(y_k)} \\ &\geq 2 + s_0 + s_1 + ... + s_{k-2} \end{aligned}$$

其中最后一行成立是因为 $rank(y_i) \geq i - 2$ 并且 s_k 单调增, 所以 $s_{rank(y_i)} \geq s_{i-2}$。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:

$$\begin{aligned} |x| &\geq s_k \\ &\geq 2 + s_0 + s_1 + ... + s_{k-2} \\ &\geq 2 + F_2 + F_3 + ... + F_k && \text{归纳假设} \\ &= 1 + F_0 + F_1 + F_2 + ... + F_k && \text{利用} F_0 = 0, F_1 = 1 \end{aligned}$$

现在, 我们需要证明

$$F_{k+2} = 1 + \sum_{i=0}^{k} F_i \tag{10.22}$$

再次使用数学归纳法:

- 边界情况: $F_2 = 1 + F_0 = 2$
- 递归情况, 假设 $k+1$ 时成立。

$$\begin{aligned} F_{k+2} &= F_{k+1} + F_k \\ &= (1 + \sum_{i=0}^{k-1} F_i) + F_k \quad \text{归纳假设} \\ &= 1 + \sum_{i=0}^{k} F_i \end{aligned}$$

综上, 我们得到最终结论:

$$n \geq |x| \geq F_{k+2} \tag{10.23}$$

□

根据斐波那契数列的性质: $F_k \geq \phi^k$, 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 $maxRank$ 为:

$$maxRank(n) = 1 + \lfloor \log_\phi n \rfloor \tag{10.24}$$

或者利用斐波那契数列的递归定义实现 MAX-RANK:

```
1: function MAX-RANK(n)
2:     F_0 ← 0, F_1 ← 1
3:     k ← 2
4:     repeat
5:         F_k ← F_{k_1} + F_{k_2}
6:         k ← k + 1
7:     until F_k < n
8:     return k − 2
```

10.4 配对堆

斐波那契堆的实现较为复杂。本节介绍配对堆。它实现简单、性能优异。大部分操作, 包括插入、获取堆顶、合并都是常数时间复杂度, 人们猜测它的弹出操作的分摊复杂度为 $O(\lg n)$ [58] [3]。

10.4.1 定义

配对堆实现为一棵多叉树。最小元素保存于树根。一个配对堆要么为空 $\varnothing$, 要么是一棵 k 叉树, 包含一个根节点和一组子树, 记为 (x, ts)。多叉树也可用“左侧孩子, 右侧兄弟”方法进行定义。

```
data PHeap a = E | Node a [PHeap a]
```

10.4.2 合并、插入、获取堆顶

合并两个配对堆时, 存在两种情况:

1. 任何一个堆为 $\varnothing$, 结果为另一个堆;
2. 否则, 比较两个堆的根节点, 把较大的一个作为另一个的新子树。

$$
\begin{aligned}
merge\ \varnothing\ h_2 &= h_2 \\
merge\ h_1\ \varnothing &= h_1 \\
merge\ (x, ts_1)\ (y, ts_2) &= \begin{cases} x < y: & (x, (y, ts_2) : ts_1) \\ \text{否则}: & (y, (x, ts1) : ts_2) \end{cases}
\end{aligned} \tag{10.25}
$$

合并的性能为常数时间。使用“左侧孩子, 右侧兄弟”方法, 我们把根节点较大的堆链接到另一个堆的子树前:

```
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:     SUB-TREES(H1) ← LINK(H2, SUB-TREES(H1))
9:     PARENT(H2) ← H1
10:    return H1
```

使用合并函数, 可以像斐波那契堆一样实现插入操作, 如式 (10.12)。堆顶元素可以从根节点获取:$top\ (x, ts) = x$。插入和获取堆顶的复杂度都是常数时间。

10.4.3 提升优先级

节点的值减小后, 我们将以它为根的子树切下, 然后合并到堆中。如果是根节点, 我们可以直接减小元素的值。

```
1: function DECREASE(H, x, k)
```

```
    KEY(x) ← k
    p ← PARENT(x)
    if p ≠ NIL then
        Remove x from SUB-TREES(p)
        PARENT(x) ← NIL
        return MERGE(H, x)
    return H
```

10.4.4 弹出

弹出堆顶的根节点后，我们将剩下的子树归并成一棵树：

$$pop\ (x, ts) = consolidate\ ts \tag{10.26}$$

我们先从左向右，两两成对地将子树合并。然后再从右向左合并合并成一棵树。配对堆的名字就来自这一合并过程。如图10.10和10.11所示。合并过程和自底向上的归并排序[3]类似。

$$\begin{aligned} consolidate\ [\,] &= \varnothing \\ consolidate\ [t] &= t \\ consolidate\ (t_1 : t_2 : ts) &= merge\ (merge\ t_1\ t_2)\ (consolidate\ ts) \end{aligned} \tag{10.27}$$

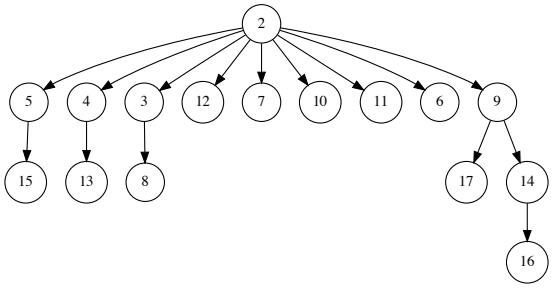
对应“左侧孩子，右侧兄弟”的实现如下：

```
function POP(H)
    L ← NIL
    for every T_x, T_y in SUB-TREES(H) do
        T ← MERGE(T_x, T_y)
        L ← LINK(T, L)
    H ← NIL
    for T in L do
        H ← MERGE(H, T)
    return H
```

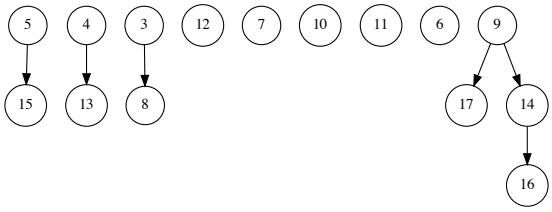
我们从左向右每次成对迭代两棵子树 T_x、T_y 合并成 T，然后链接到 L 的前面。这样再次遍历 L 时，实际是按照从右向左的顺序。堆中可能含有奇数棵子树，这种情况下，最后一次 $T_y = \text{NIL}$，成对合并后 $T = T_x$。

10.4.5 删除

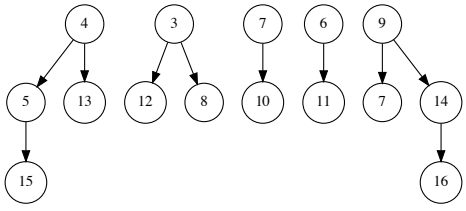
为了删除某个节点 x，我们可以先将节点的值减小为 $-\infty$，然后再执行一次弹出操作。本节介绍另外一种删除方法。若 x 为根节点，我们只需要执行一次弹出操作。否则，我们将 x 从堆 H 中切下，然后对 x 执行一次弹出操作，再将结果合并回 H：



(a) 弹出前的配对堆

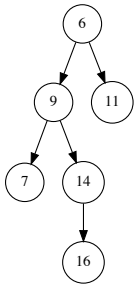


(b) 根节点 2 被删除, 剩余 9 棵子树

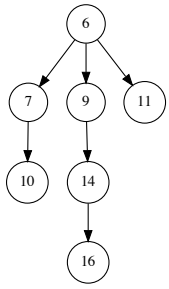


(c) 每两棵树成对合并,因为有奇数棵树,
所以最后一棵无需合并。

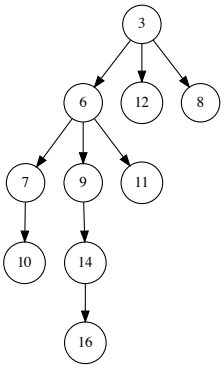
图 10.10: 删除根节点, 将子树成对合并



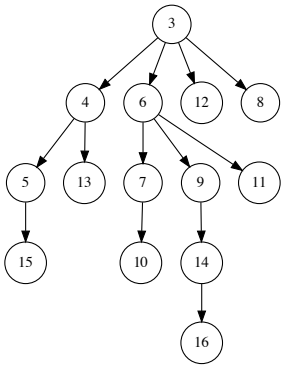
(a) 将根节点为 9 和 6 的两棵树合并



(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 10.11: 从右向左合并的步骤

```
1: function DELETE(H, x)
2:     if H = x then
3:         POP(H)
4:     else
5:         H ← CUT(H, x)
6:         x ← POP(x)
7:         MERGE(H, x)
```

因为删除算法调用弹出操作, 我们猜想它的分摊性能也是对数时间 $O(\lg n)$。

练习 10.3

实现配对堆的删除。

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多叉树森林作为底层数据结构, 而配对堆实现为一棵多叉树。通过将某些耗时的操作延迟进行, 可以获得总体上优异的分摊性能。这一点很具有启发性。

10.6 附录:例子程序

多叉树的定义(左侧孩子, 右侧兄弟):

```
data Node<K> {
    Int rank
    K key
    Node<K> parent, subTrees, sibling,
    Bool mark

    Node(K x) {
        key = x
        rank = 0
        parent = subTrees = sibling = null
        mark = false
    }
}
```

二项式树链接:

```
Node<K> link(Node<K> t1, Node<K> t2) {
    if t2.key < t1.key then (t1, t2) = (t2, t1)
    t2.sibling = t1.subTrees
    t1.subTrees = t2
    t2.parent = t1
    t1.rank = t1.rank + 1
    return t1
```

```
}
```

二项式堆插入:

```
Node<K> insert(K x, Node<K> h) = insertTree(Node(x), h)

Node<K> insertTree(Node<K> t, Node<K> h) {
    var h1 = Node()
    var prev = h1
    while h ≠ null and h.rank ≤ t.rank {
        var t1 = h
        h = h.sibling
        if t.rank == t1.rank {
            t = link(t, t1)
        } else {
            prev.sibling = t1
            prev = t1
        }
    }
    prev.sibling = t
    t.sibling = h
    return removeFirst(h1)
}

Node<K> removeFirst(Node<K> h) {
    var next = h.sibling
    h.sibling = null
    return next
}
```

二项式堆插入的递归实现:

```
data BiTree a = Node { rank :: Int
                     , key :: a
                     , subTrees :: [BiTree a]}

type BiHeap a = [BiTree a]

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)

insertTree t [] = [t]
insertTree t ts@(t':ts') | rank t < rank t' = t:ts
                         | rank t > rank t' = t' : insertTree t ts'
                         | otherwise = insertTree (link t t') ts'

insert x = insertTree (Node 0 x [])
```

二项式堆的合并:

```
Node<K> merge(h1, h2) {
    var h = Node()
    var prev = h
```

```
    while h1 ≠ null and h2 ≠ null {
        if h1.rank < h2.rank {
            prev.sibling = h1
            prev = prev.sibling
            h1 = h1.sibling
        } else if h2.rank < h1.rank {
            prev.sibling = h2
            prev = prev.sibling
            h2 = h2.sibling
        } else {
            var (t1, t2) = (h1, h2)
            (h1, h2) = (h1.sibling, h2.sibling)
            h1 = insertTree(link(t1, t2), h1)
    }
    if h1 ≠ null then prev.sibling = h1
    if h2 ≠ null then prev.sibling = h2
    return removeFirst(h)
}
```

递归合并两个二项式堆:

```
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 (link t1 t2) (merge ts1' ts2')
```

二项式堆的弹出

```
Node<K> reverse(Node<K> h) {
    Node<K> prev = null
    while h ≠ null {
        var x = h
        h = h.sibling
        x.sibling = prev
        prev = x
    }
    return prev
}

(Node<K>, Node<K>) extractMin(Node<K> h) {
    var head = h
    Node<K> tp = null
    Node<K> tm = null
    Node<K> prev = null
    while h ≠ null {
        if tm == null or h.key < tm.key {
            tm = h
            tp = prev
        }
        prev = h
        h = h.sibling
    }
```

```
    if tp ≠ null {
        tp.sibling = tm.sibling
    } else {
        head = tm.sibling
    }
    tm.sibling = null
    return (tm, head)
}

(K, Node<K>) pop(Node<K> h) {
    var (tm, h) = extractMin(h)
    h = merge(h, reverse(tm.subtrees))
    tm.subtrees = null
    return (tm.key, h)
}
```

二项式堆弹出的递归实现:

```
pop h = merge (reverse $ subTrees t) ts where
    (t, ts) = extractMin h

extractMin [t] = (t, [])
extractMin (t:ts) = if key t < key t' then (t, ts)
                    else (t', t:ts') where
    (t', ts') = extractMin ts
```

使用双向链表合并斐波那契堆:

```
FibHeap<K> merge(FibHeap<K> h1, FibHeap<K> h2) {
    if isEmpty(h1) then return h2
    if isEmpty(h2) then return h1
    FibHeap<K> h = FibHeap<K>()
    h.trees = concat(h1.trees, h2.trees)
    h.minTree = if h1.minTree.key < h2.minTree.key
                then h1.minTree else h2.minTree
    h.size = h1.size + h2.size
    return h
}

bool isEmpty(FibHeap<K> h) = (h == null or h.trees == null)

Node<K> concat(Node<K> first1, Node<K> first2) {
  var last1 = first1.prev
  var last2 = first2.prev
  last1.next = first2
  first2.prev = last1
  last2.next = first1
  first1.prev = last2
  return first1
}
```

斐波那契树归并:

```
consolidate = foldr melt [] where
```

```
    melt t [] = [t]
    meld t (t':ts) | rank t == rank t' = meld (link t t') ts
                   | rank t <  rank t' = t : t' : ts
                   | otherwise = t' : meld t ts
```

使用辅助数组进行归并:

```
void consolidate(FibHeap<K> h) {
    Int R = maxRank(h.size) + 1
    Node<K>[R] a = [null, ...]
    while h.trees ≠ null {
        var x = h.trees
        h.trees = remove(h.trees, x)
        Int r = x.rank
        while a[r] ≠ null {
            var y = a[r]
            x = link(x, y)
            a[r] = null
            r = r + 1
        }
        a[r] = x
    }
    h.minTr = null
    h.trees = null
    for var t in a if t ≠ null {
        h.trees = append(h.trees, t)
        if h.minTr == null or t.key < h.minTr.key then h.minTr = t
    }
}
```

斐波那契堆的弹出:

```
pop (FH _ (Node _ x []) []) = (x, E)
pop (FH sz (Node _ x tsm) ts) = (x, FH (sz - 1) tm ts') where
    (tm, ts') = extractMin $ consolidate (tsm ++ ts)
```

提升优先级:

```
void decrease(FibHeap<K> h, Node<K> x, K k) {
    var p = x.parent
    x.key = k
    if p ≠ null and k < p.key {
        cut(h, x)
        cascadeCut(h, p)
    }
    if k < h.minTr.key then h.minTr = x
}

void cut(FibHeap<K> h, Node<K> x) {
    var p = x.parent
    p.subTrees = remove(p.subTrees, x)
    p.rank = p.rank - 1
    h.trees = append(h.trees, x)
    x.parent = null
```

```
    x.mark = false
}

void cascadeCut(FibHeap<K> h, Node<K> x) {
    var p = x.parent
    if p == null then return
    if x.mark {
        cut(h, x)
        cascadeCut(h, p)
    } else {
        x.mark = true
    }
}
```

第十一章　队列

11.1　简介

队列提供了先进先出（FIFO）的机制。可以用多种方法实现队列，例如单向、双向链表，循环缓冲区等，Okasaki 给出了 16 种不同的实现方法[3]。队列需要满足下面的两条基本要求：

1. 可以在常数时间内向末尾添加元素；
2. 可以在常数时间内从头部获取或删除元素。

可以用双向链表直观地实现队列。我们略去这个简单的实现，而关注如何用其它基本数据结构，如列表、数组实现队列。

11.2　列表实现

我们可以用常数时间在列表头部插入、删除元素。但为了先进先出，我们只能在头部执行一种操作，而在尾部执行另一种操作。我们需要 $O(n)$ 时间遍历整个列表以到达尾部，其中 n 是列表长度。这样就无法达到性能要求。为了解决这个问题，可以一个变量记录尾部位置。并用一个额外的节点 S 简化空队列的处理，如图11.1所示。

```
data Node<K> {
  Key key
  Node next
}

data Queue {
  Node head, tail
}
```

队列中最基本的两个操作是入队（Enqueue，或 push、snoc、append、push back）和出队（Dequeue，或 pop、pop front）。使用列表时，我们选择在头部加入元素、从尾部删除元素以简化实现。

1: **function** ENQUEUE(Q, x)
2: $p \leftarrow$ NODE(x)

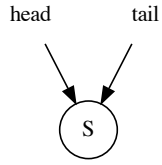


图 11.1: 空队列, 头、尾都指向 S

3: $\text{NEXT}(p) \leftarrow \text{NIL}$
4: $\text{NEXT}(\text{TAIL}(Q)) \leftarrow p$
5: $\text{TAIL}(Q) \leftarrow p$

队列至少有一个节点(空队列中有 S 节点), 因此无需检查尾部是否为 NIL。

1: **function** DEQUEUE(Q)
2: $x \leftarrow \text{HEAD}(Q)$
3: $\text{NEXT}(\text{HEAD}(Q)) \leftarrow \text{NEXT}(x)$
4: **if** $x = \text{TAIL}(Q)$ **then** ▷ Q 变为空
5: $\text{TAIL}(Q) \leftarrow \text{HEAD}(Q)$
6: **return** KEY(x)

S 节点在所有其它节点的前面, HEAD 实际返回 S 的下一个节点, 如图11.2所示。我们可以把这一实现扩展到并发环境。在头部和尾部各使用一把并发锁。S 节点可以在队列空时避免死锁[59]、[60]。

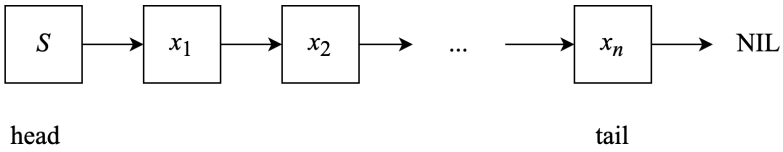


图 11.2: 带有 S 节点的列表

11.3 循环缓冲区

和列表相反, 我们可以在常数时间将元素添加到数组末尾, 但需要线性时间 $O(n)$ 从头部删除。这是因为要将全部剩余元素依次向前移动。为了达到队列的性能要求, 我们可以把数组的头尾连接起来, 做成一个环, 叫做循环缓冲区, 如图如图11.3、11.4所示。这样用数组的头部坐标 head, 队列长度 count, 和数组大小 size, 就可以完全表述队列。count 等于 0 时队列为空, 等于 size 时队列已满。我们还可以利用模运算简化入队、出队的实现。

1: **function** ENQUEUE(Q, x)
2: **if** not FULL(Q) **then**

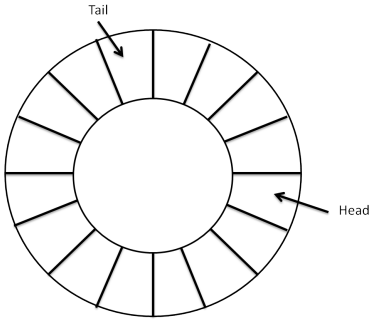
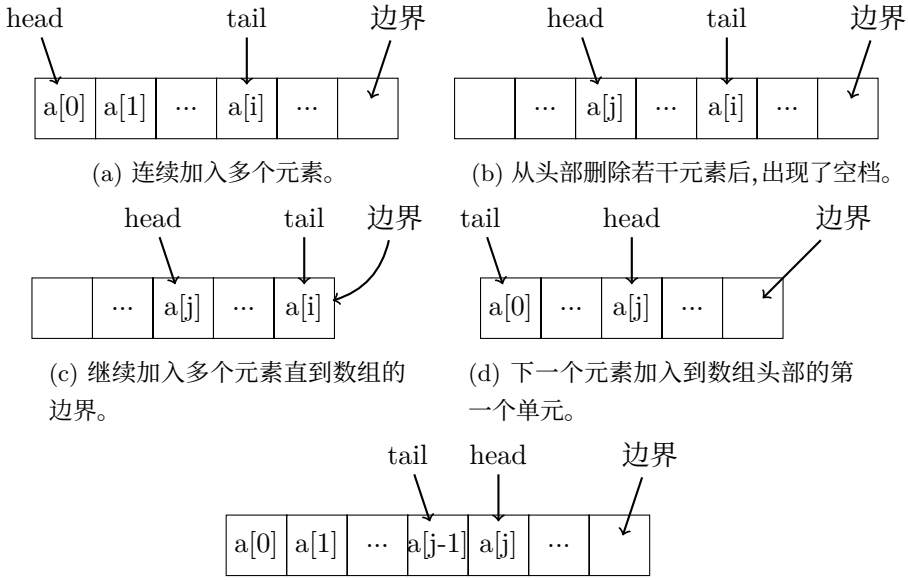


图 11.3: 循环缓冲区



(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 11.4: 使用循环缓冲区实现队列

```
3:        Count(Q) ← Count(Q) + 1
4:        tail ← (Head(Q) + Count(Q)) mod Size(Q)
5:        Buf(Q)[tail] ← x

1: function Dequeue(Q)
2:     x ← NIL
3:     if not Empty(Q) then
4:         h ← Head(Q)
5:         x ← Buf(Q)[h]
6:         Head(Q) ← (h + 1) mod Size(Q)
7:         Count(Q) ← Count(Q) - 1
8:     return x
```

练习 11.1

循环缓冲区在初始化时规定了最大的容量，如果使用头、尾两个指针，而不用 Count，如何检测队列是否为空？是否已满？（考虑两种情况：头部在尾部前面，和头部在尾部后面）。

11.4 双列表队列

列表的头部操作为常数时间，但尾部需要线性时间。我么可以把两个列表“尾对尾”连起来实现队列。形状类似一个马蹄形磁铁，如图11.5所示。两个列表分别叫做前（front）和后（rear）。队列记为 (f, r)，空队列等于 $([\,], [\,])$。我们把新元素加入 r 的头部，出队时，将元素从 f 的头部取走，性能都是常数时间。

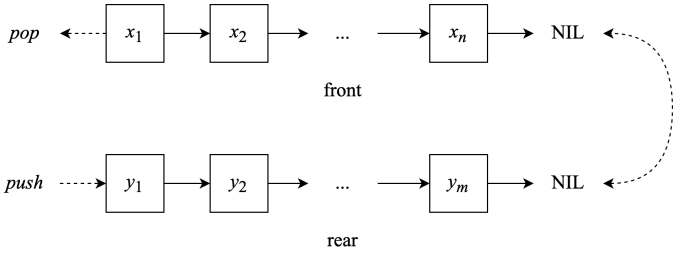


图 11.5: 双列表队列

$$\begin{cases} push\ x\ (f, r) & = (f, x{:}r) \\ pop\ (x{:}f, r) & = (f, r) \end{cases} \tag{11.1}$$

经过一系列出队操作后，f 可能为空，而 r 中还有元素。为了能继续出队，我们将 r 反转后替换掉 f，即：$([\,], r) \mapsto (reverse\ r, [\,])$。为此每次出入队后，需要执行一次平

衡检查和调整:

$$\begin{array}{rcl} balance\ [\,]\ r & = & (reverse\ r, [\,]) \\ balance\ f\ r & = & (f, r) \end{array} \tag{11.2}$$

一旦发生 r 的反转,则这次操作的性能下降为线性时间。尽管如此,整体的分摊复杂度是常数时间的。我们重新定义入队和出队为:

$$\begin{cases} push\ x\ (f, r) & = balance\ f\ (x{:}r) \\ pop\ (x{:}f, r) & = balance\ f\ r \end{cases} \tag{11.3}$$

我们可以用数组给出一个双列表的对称实现。利用表11.1的对称性, 我们将两个数组“头对头”连接起来形成队列, 如图11.6所示。当 R 数组为空时, 我们将 F 数组反转替换掉 R 数组。

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

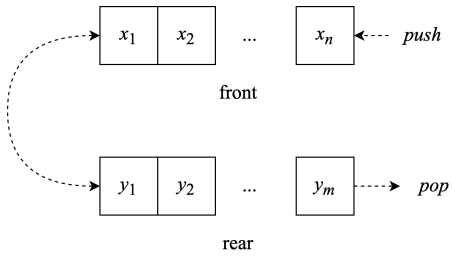


图 11.6: 双数组队列

练习 11.2

1. 为什么要在 push 时也要进行平衡检查和调整?
2. 证明双列表队列的分摊复杂度为常数时间。
3. 实现双数组队列。

11.5 平衡队列

虽然双列表队列的分摊复杂度为常数时间, 但最坏情况下的性能是线性的。例如 f 中有一个元素, 此后连续将 n 个元素加入队列, 此时执行出队的复杂度为 $O(n)$。这

一问题的原因是 f 和 r 的长度不平衡。为了改进平衡性，我们加入一条规则，要求 r 的长度不大于 f 的长度，否则就反转列表。

$$|r| \leq |f| \tag{11.4}$$

每次操作都需要检查长度，但这需要线性时间。为此我们将长度记录下来，并在出入队时更新。这样双列表队列就表示为 (f, n, r, m)，其中 $n = |f|, m = |r|$，分别是两个列表的长度。根据平衡规则 (11.4)，我们可以检查 f 的长度来判断队列是否为空：

$$Q = \phi \iff n = 0 \tag{11.5}$$

我们更新出、入队的定义为：

$$\begin{cases} push\ x\ (f, n, r, m) & = balance\ (f, n, x{:}r, m + 1) \\ pop\ (x{:}f, n, r, m) & = balance\ (f, n - 1, r, m) \end{cases} \tag{11.6}$$

其中 $balance$ 定义为：

$$balance\ (f, n, r, m) = \begin{cases} m \leq n: & (f, n, r, m) \\ \text{否则}: & (f +\!\!+ reverse\ r, m + n, [\,], 0) \end{cases} \tag{11.7}$$

11.6 实时队列

在平衡队列的实现中，列表连接、反转的性能仍然是线性时间的。在实时系统中，需要进一步改进。性能瓶颈出现在 $f +\!\!+ reverse\ r$ 中。此时 $m > n$，违反了平衡规则。由于 m、n 都是整数，我们进一步知道：$m = n + 1$。$+\!\!+$ 的复杂度是 $O(n)$，反转操作的复杂度是 $O(m)$，总复杂度是 $O(n + m)$，和队列中元素个数成正比。我们可以将这一操作分派到各次出、入队中去。首先分析一下尾递归的反转操作：

$$reverse\ = reverse'\ [\,] \tag{11.8}$$

这一定义是克里化的，其中：

$$\begin{aligned} reverse'\ a\ [\,] &= a \\ reverse'\ a\ (x{:}xs) &= reverse'\ (x{:}a)\ xs \end{aligned} \tag{11.9}$$

可以很容易地将尾递归[61] [62] 定义转换为逐步计算。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）。接下来我们利用状态机调度调度（slow-down）反转计算：

$$\begin{aligned} step\ S_r\ a\ [\,] &= (S_f, a) \\ step\ S_r\ a\ (x{:}xs) &= (S_r, (x{:}a), xs) \end{aligned} \tag{11.10}$$

每一步, 我们先检查当前的状态, 如果为 S_r(反转中), 但是列表中已没有剩余元素需要反转, 就将状态变为完成 S_f; 否则, 我们取出列表中的第一个元素 x, 将其链结到 a 的前面。接下来我们不再进行递归, 这一步计算到此结束。当前的状态和反转的中间结果被保存下来, 供以后再次调用 $step$ 时使用。例如:

$$\begin{aligned}
step\ S_r\ \text{“hello”}\ [\,] &= (S_r, \text{“ello”}, \text{“h”}) \\
step\ S_r\ \text{“ello”}\ \text{“h”} &= (S_r, \text{“llo”}, \text{“eh”}) \\
&\cdots \\
step\ S_r\ \text{“o”}\ \text{“lleh”} &= (S_r, [\,], \text{“olleh”}) \\
step\ S_r\ [\,]\ \text{“olleh”} &= (S_f, \text{“olleh”})
\end{aligned}$$

现在我们可以将反转计算逐步分派到出、入队中。但是这仅解决了一半问题。我们需要逐步分解、调度 $+\!\!+$。实现逐步连接的难度更大。我们利用逐步反转的结果, 并使用一个技巧: 为了实现 $xs +\!\!+ ys$, 我们可以先将 xs 反转为 $\overleftarrow{xs}$, 然后逐一将 $\overleftarrow{xs}$ 中的元素取出, 放到 ys 的前面。这和 $reverse'$ 类似。

$$\begin{aligned}
xs +\!\!+ ys &= (reverse\ reverse\ xs) +\!\!+ ys \\
&= (reverse'\ [\,]\ (reverse\ xs)) +\!\!+ ys \\
&= reverse'\ ys\ (reverse\ xs) \\
&= reverse'\ ys\ \overleftarrow{xs}
\end{aligned} \tag{11.11}$$

这一事实表明, 我们可以增加另一个状态来控制 $step$, 在 r 反转后, 逐步操作 $\overleftarrow{f}$ 实现连接。三种状态为: 反转 S_r、连接 S_c、完成 S_f。整个操作被分解为两个阶段:

1. 同时反转 f 和 r, 逐步得到 $\overleftarrow{f}$ 和 $\overleftarrow{r}$;
2. 逐步从 $\overleftarrow{f}$ 取出元素, 链接到 $\overleftarrow{r}$ 前面。

$$\begin{aligned}
next\ (S_r, f', x{:}f, r', y{:}r) &= (S_r, x{:}f', f, y{:}r', r) && \text{同时反转} f, r \\
next\ (S_r, f', [\,], r', [y]) &= next\ (S_c, f', y{:}r') && \text{反转结束、转入连接} \\
next\ (S_c, a, [\,]) &= (S_f, a) && \text{连接结束} \\
next\ (S_c, a, x{:}f') &= (S_c, x{:}a, f') && \text{逐步连接}
\end{aligned} \tag{11.12}$$

接下来我们需要将这些递进的步骤分配到每个出、入队操作中以实现实时队列。根据平衡队列的条件, 当 $m = n + 1$ 时, 我们开始逐步计算 $f +\!\!+ reverse\ r$。总共需要 $n + 1$ 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 $n + 1$ 步来进行连接操作。因此总共花费了 $2n + 2$ 步。最直接的想法是在每一个出、入队中分配一个递进步骤。但这里有一个关键问题: 在完成 $2n + 2$ 步操作之前, 队列有没有可能由于接下来的一系列出、入队操作再次变得不平衡?

幸运的是, 在花费 $2n + 2$ 步完成 $f +\!\!+ reverse\ r$ 之前, 连续的入队操作不可能再次使队列变得不平衡。一旦开始恢复平衡的处理, 经过 $2n + 2$ 步后, 我们就得到了一

个新的 f 列表 $f' = f +\!\!+ reverse\ r$。而下一次队列变得不平衡时有：

$$\begin{aligned} |r'| &= |f'| + 1 \\ &= |f| + |r| + 1 \\ &= 2n + 2 \end{aligned} \tag{11.13}$$

也就是说，从上次不平衡的时刻算起，即使不断持续入队新元素，以最快的速度再次使得队列不平衡时，$2n+2$ 步计算恰好已经完成了。此时新的 f 列表被计算出来。我们可以安全地继续计算 $f' +\!\!+ reverse\ r'$。多亏了平衡规则，帮助我们保证了这一点。

但不幸的是，在 $2n+2$ 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 f 列表取出元素，但是新的 f 列表 $f' = f +\!\!+ reverse\ r$ 尚未计算好。此时没有一个可用的 f 列表。为了解决这个问题，我们在第一阶段并行计算 $reverse\ f$ 时，另外保存一份 f 的副本。这样即使连续进行 n 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[1]。

保存的 f 副本	进行中的计算	新的 r 列表
$\{f_i, f_{i+1}, ..., f_n\}$	$(S_r, \tilde{f}, ..., \tilde{r}, ...)$	$\{...\}$
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 n 步完成之前的队列中间状态

经过 n 次出队操作，f 的副本已经用光。我们此时刚刚开始逐步连接的计算阶段。此时如果继续出队会怎样？事实上，由于 f 的副本被用光，变成了 $[\,]$，我们无需再进行连接操作了。这是因为 $f +\!\!+ \overleftarrow{r} = [\,] +\!\!+ \overleftarrow{r} = \overleftarrow{r}$。事实上，在进行连接操作时，我们只需要将 f 中尚未出队的部分连接起来。因为元素从 f 的头部逐一出队，我们可以使用一个计数器来记录 f 中剩余元素的个数。当开始计算 $f +\!\!+ reverse\ r$ 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。下面是增加了计数器的状态转换定义：

$$\begin{aligned} next\ (S_r, n, f', x{:}f, r', y{:}r) &= (S_r, n+1, x{:}f', f, y{:}r', r) && \text{同时反转} f, r \\ next\ (S_r, n, f', [\,], r', [y]) &= next\ (S_c, n, f', y{:}r') && \text{反转结束、转入连接} \\ next\ (S_c, 0, a, f) &= (S_f, a) && \text{连接结束} \\ next\ (S_c, n, a, x{:}f') &= (S_c, n-1, x{:}a, f') && \text{逐步连接} \\ next\ S_0 &= S_0 && \text{空闲状态} \end{aligned} \tag{11.14}$$

我们还定义了一个空闲状态 S_0 来简化状态转换的实现。队列的数据结构分为三个部分：f 列表及其长度 n、正在计算中的 $f +\!\!+ reverse\ r$ 的中间状态、r 列表及其长度

[1] 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。我们复制的是 f 列表的引用。每个元素的复制被推迟到后续各个步骤中。

m。记为 (f, n, S, r, m)。空队列记为 $([\,], 0, S_0, [\,], 0)$。根据平衡规则当 $n = 0$ 时队列为空。我们修改出、入队定义为：

$$\begin{cases} push\ x\ (f, n, S, r, m) & = balance\ f\ n\ S\ (x{:}r)\ (m+1) \\ pop\ (x{:}f, n, S, r, m) & = balance\ f\ (n-1)\ (abort\ S)\ r\ m \end{cases} \tag{11.15}$$

其中 $abort$ 在出队时递减计数器，这样将来可以少连接一个元素。我们稍后定义这一撤销操作。$balance$ 检查平衡规则，若不满足则启动 $f +\!\!+ reverse\ r$ 逐步恢复平衡；否则执行一步尚未完成的递进计算：

$$balance\ f\ n\ S\ r\ m = \begin{cases} m \le n: & step\ f\ n\ S\ r\ m \\ \text{否则}: & step\ f\ (n+m)\ (next\ (S_r, 0, [\,], f, [\,], r))\ [\,]\ 0 \end{cases} \tag{11.16}$$

其中 $step$ 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。

$$step\ f\ n\ S\ r\ m = queue\ (next\ S) \tag{11.17}$$

其中：

$$\begin{array}{rcll} queue\ (S_f, f') & = & (f', n, S_0, r, m) & \text{用逐步计算结果}f'\text{替换}f \\ queue\ S' & = & (f, n, S', r, m) & \end{array} \tag{11.18}$$

我们还需要实现 $abort$ 函数，指示状态机，由于发生了出队操作，可以少连接一个元素。

$$\begin{array}{rcl} abort\ (S_c, 0, (x{:}a), f') & = & (S_f, a) \\ abort\ (S_c, n, a, f') & = & (S_c, n-1, a, f') \\ abort\ (S_r, n, f'f, r'r) & = & (S_r, n-1, f', f, r', r) \\ abort\ S & = & S \end{array} \tag{11.19}$$

练习 11.3

1. 在 $abort$ 函数中，当 $n = 0$ 时，当 $n = 0$ 时，我们实际上撤销了上一个链接元素的操作，去掉了 x 而返回 a 作为结果。为什么需要回滚一个元素？
2. 使用双数组实现实时队列。注意：当开始递进反转时，不能一次性复制数组，否则就会将性能降低到线性时间。请实现一个惰性复制，使得每步反转时仅复制一个元素。

11.7 惰性实时队列

实时队列的关键在于将耗时的 $f +\!\!+ reverse\ r$ 计算分解。利用惰性求值可以得到一个简化的实现。假设函数 $rotate$ 可以逐步计算 $f +\!\!+ reverse\ r$。也就是说，使用一个

累积器 a, 下面的两个函数等价:

$$rotate\ xs\ ys\ a = xs \mathbin{+\!\!+} (reverse\ ys) \mathbin{+\!\!+} a \tag{11.20}$$

我们将 xs 初始化为 f 列表, ys 初始化为 r 列表, a 初始化为空 [ ]。为了实现轮转, 我们先考虑边界情况:

$$rotate\ [\,]\ [y]\ a = y{:}a \tag{11.21}$$

递归情况为:

$$\begin{array}{lll} & rotate\ (x{:}xs)\ (y{:}ys)\ a & \\ = & (x{:}xs) \mathbin{+\!\!+} (reverse\ (y{:}ys)) \mathbin{+\!\!+} a & \text{定义 (11.20)} \\ = & x : (xs \mathbin{+\!\!+} reverse\ (y{:}ys)) \mathbin{+\!\!+} a) & \text{连接的结合性} \\ = & x : (xs \mathbin{+\!\!+} reverse\ ys \mathbin{+\!\!+} (y{:}a)) & \text{反转的性质和连接的结合性} \\ = & x : rotate\ xs\ ys\ (y{:}a) & \text{反向用定义 (11.20)} \end{array} \tag{11.22}$$

归纳上面的两种情况, 可以得到最终的轮转算法:

$$\begin{array}{rcl} rotate\ [\,]\ [y]\ a & = & y{:}a \\ rotate\ (x{:}xs)\ (y{:}ys)\ a & = & x : rotate\ xs\ ys\ (y{:}a) \end{array} \tag{11.23}$$

在惰性执行环境中, (:) 操作会推迟到出、入队时才执行, 这样就将 $rotate$ 计算自然分摊了。为此我们修改双列表的定义为 (f, r, rot), 其中 rot 表示正在进行的的轮转计算 $f \mathbin{+\!\!+} reverse\ r$, 它初始为空 [ ]。

$$\begin{cases} push\ x\ (f, r, rot) & = balance\ f\ (x{:}r)\ rot \\ pop\ (x{:}f, r, rot) & = balance\ f\ r\ rot \end{cases} \tag{11.24}$$

每次 $balance$ 操作都会向前推进一次轮转计算, 当轮转结束时, 我们开始新一轮计算:

$$\begin{array}{rcll} balance\ f\ r\ [\,] & = & (f', [\,], f') & \text{其中}: f' = rotate\ f\ r\ [\,] \\ balance\ f\ r\ (x{:}rot) & = & (f, r, rot) & \text{推进轮转} \end{array} \tag{11.25}$$

练习 11.4

如何实现双向队列, 在头部尾部都支持常数时间的元素添加和删除。

11.8 附录:例子程序

列表实现的出、入队:

```
Queue<K> enQ(Queue<K> q, K x) {
    var p = Node(x)
    p.next = null
    q.tail.next = p
    q.tail = p
    return q
}

K deQ(Queue<K> q) {
    var p = q.head.next   //the next of S
    q.head.next = p.next
    if q.tail == p then q.tail = q.head //empty
    return p.key
}
```

循环缓冲区的定义:

```
data Queue<K> {
    K buf[]
    int head, cnt, size

    Queue(int max) {
        buf = Array<K>(max)
        size = max
        head = cnt = 0
    }
}
```

使用循环缓冲区的出、入队:

```
N offset(N i, N size) = if i < size then i else i - size

void enQ(Queue<K> q, K x) {
    if q.cnt < q.size {
        q.buf[offset(q.head + q.cnt, q.size)] = x;
        q.cnt = q.cnt + 1
    }
}

K head(Queue<K> q) = if q.cnt == 0 then null else q.buf[q.head]

K deQ(Queue<K> q) {
    K x = null
    if q.cnt > 0 {
        x = head(q)
        q.head = offset(q→head + 1, q→size);
        q.cnt = q.cnt -1
    }
    return x
}
```

实时队列

```
data State a = Empty
             | Reverse Int [a] [a] [a] [a] -- n, acc f, f, acc r, r
             | Concat Int [a] [a]          -- n, acc, reversed f
             | Done [a]  -- f' = f ++ reverse r

-- f, n = length f, state, r, m = length r
data RealtimeQueue a = RTQ [a] Int (State a) [a] Int

push x (RTQ f n s r m) = balance f n s (x:r) (m + 1)

pop (RTQ (_:f) n s r m) = balance f (n - 1) (abort s) r m

top (RTQ (x:_) _ _ _ _) = x

balance f n s r m
    | m ≤ n =  step f n s r m
    | otherwise = step f (m + n) (next (Reverse 0 [] f [] r)) [] 0

step f n s r m = queue (next s) where
  queue (Done f') = RTQ f' n Empty r m
  queue s' = RTQ f n s' r m

next (Reverse n f' (x:f) r' (y:r)) = Reverse (n + 1) (x:f') f (y:r') r
next (Reverse n f' [] r' [y]) = next $ Concat n (y:r') f'
next (Concat 0 acc _) = Done acc
next (Concat n acc (x:f')) = Concat (n-1) (x:acc) f'
next s = s

abort (Concat 0 (_:acc) _) = Done acc -- rollback 1 elem
abort (Concat n acc f') = Concat (n - 1) acc f'
abort (Reverse n f' f r' r) = Reverse (n - 1) f' f r' r
abort s = s
```

惰性实时队列:

```
data LazyRTQueue a = LQ [a] [a] [a] -- front, rear, f ++ reverse r

empty = LQ [] [] []

push (LQ f r rot) x = balance f (x:r) rot

pop (LQ (_:f) r rot) = balance f r rot

top (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)
```

第十二章　序列

12.1　简介

序列是对数组和列表的一种抽象组合。我们希望理想的序列能达到下面的要求：

1. 可以在头部、尾部以常数时间插入、删除元素；
2. 可以快速（优于线性时间）连接两个序列；
3. 可以快速随机访问、更改任何元素；
4. 可以快速在指定位置断开序列。

数组、列表仅部分满足这些要求，如下表所示。其中 n 为单个序列的长度，n_1、n_2 分别表示被连接的两个序列的长度。

操作	数组	列表
在头部插入、删除	$O(n)$	$O(1)$
在尾部插入、删除	$O(1)$	$O(n)$
连接	$O(n_2)$	$O(n_1)$
随机访问位置 i	$O(1)$	$O(i)$
在位置 i 删除	$O(n-i)$	$O(1)$

本章我们给出三种序列实现：二叉随机访问列表、可连接列表、手指树。

12.2　二叉随机访问列表

二叉随机访问列表是由二叉树森林实现的随机访问列表。森林包含若干完全二叉树。元素只保存在叶子节点中。对任何非负整数 n，将其表达为二进制，我们就知道需要多少棵完全二叉树来存储 n 个元素。每个值为 1 的二进制位代表一棵二叉树，树的大小对应着二进制位的高低。任给索引 $1 \le i \le n$，我们都可以快速在森林中定位到保存第 i 个节点的二叉树。如图12.1所示，树 t_1、t_2 表示序列 $[x_1, x_2, x_3, x_4, x_5, x_6]$。

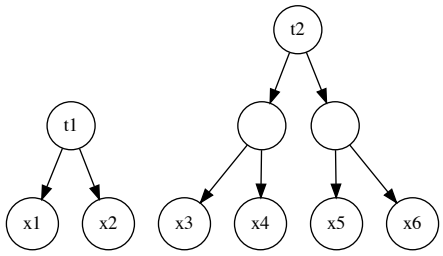


图 12.1: 含有 6 个元素的序列

记深度为 $i+1$ 的完全二叉树为 t_i。t_0 只含有一个叶子节点。t_i 含有 2^i 个叶子。对于 n 个元素的序列, 我们把 n 表示为二进制数 $n = (e_m e_{m-1} ... e_1 e_0)_2$, 其中 e_i 为 1 或 0。

$$n = 2^0 e_0 + 2^1 e_1 + ... + 2^m e_m \tag{12.1}$$

如果 $e_i \neq 0$, 就需要一棵大小为 2^i 的完全二叉树 t_i。在图12.1的例子中, 序列长度为 $6 = (110)_2$。最低位是 0, 我们不需要大小为 1 的树; 第 2 位是 1, 需要一棵大小为 2 的树 t_1; 最高位是 1, 需要一棵大小为 4 的树 t_2。这样就把序列 $[x_1, x_2, ..., x_n]$ 表示为树的列表。列表中每棵树的大小都是唯一的, 并按照从小到大排列。我们称之为**二叉随机访问列表**[3]。我们可以在二叉树定义的基础上稍作变化以实现这种列表: 1、元素只保存在叶子节点中, 2、在每棵子树中记录树的大小。这样每个分枝节点记为 (s, l, r), 其中 s 表示子树的大小, l、r 分别表示左右子树。包含元素 x 的叶子节点记为 (x)。我们可以这样获取一棵树的大小:

$$\begin{aligned} size\ (x) &= 1 \\ size\ (s, l, r) &= s \end{aligned} \tag{12.2}$$

为了把新元素 y 插入到序列 S 的前面, 我们创建一棵只有一个叶子节点的 t_0 树: $t' = (y)$, 然后把它插入到森林中。$insert\ y\ S = insert_T\ (y)\ S$, 或写成克里化形式:

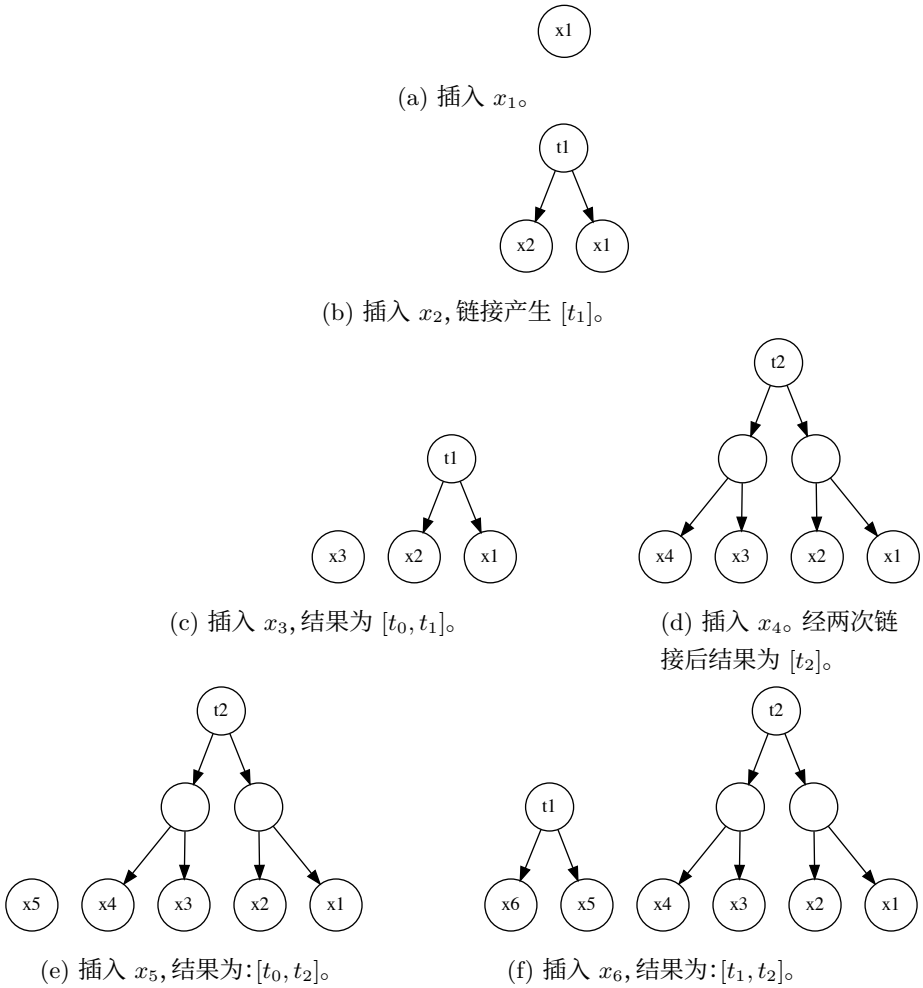
$$insert\ y = insert_T\ (y) \tag{12.3}$$

我们检查森林中的第一棵树 t_i, 比较 t_i 和 t' 的大小, 如果 t_i 较大, 就将 t' 置于森林最前面(常数时间); 若 t_i 和 t' 相等, 我们将它们链接(常数时间)成一棵较大的树: $t'_{i+1} = (2s, t_i, t')$, 然后递归地将 t'_{i+1} 插入到森林中。如图12.2所示。

$$\begin{aligned} insert_T\ t\ [\,] &= [t] \\ insert_T\ t\ (t_1 : ts) &= \begin{cases} size\ t < size\ t_1 : & t : t_1 : ts \\ \text{否则}: & insert_T\ (link\ t\ t_1)\ ts \end{cases} \end{aligned} \tag{12.4}$$

其中 $link$ 将两棵大小相同的树链接起来: $link\ t_1\ t_2 = (size\ t_1 + size\ t_2, t_1, t_2)$。

若森林中包含 m 棵树, m 的大小为 $O(\lg n)$, 头部插入的性能为 $O(\lg n)$。稍后我们证明分摊性能为常数时间。



(a) 插入 x_1。

(b) 插入 x_2，链接产生 $[t_1]$。

(c) 插入 x_3，结果为 $[t_0, t_1]$。

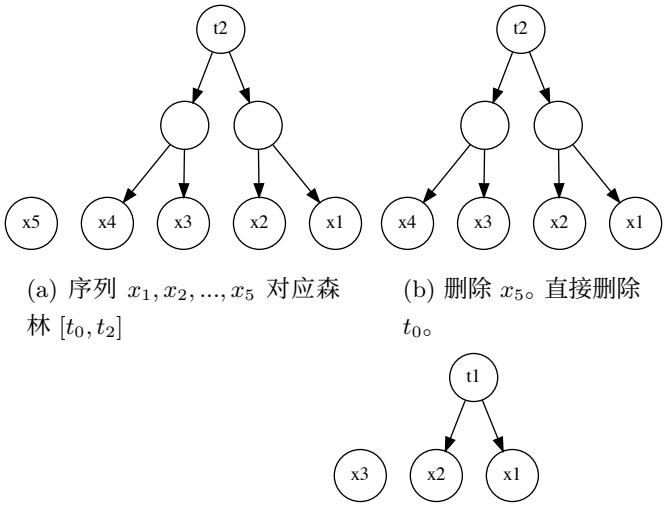
(d) 插入 x_4。经两次链接后结果为 $[t_2]$。

(e) 插入 x_5，结果为：$[t_0, t_2]$。

(f) 插入 x_6，结果为：$[t_1, t_2]$。

图 12.2: 插入 $x_1, x_2, ..., x_6$

对称地, 我们利用插入的逆过程实现从序列头部删除元素。如果森林中第一棵树是 t_0(单叶子节点), 我们直接将 t_0 删除; 否则, 递归地将第一棵树拆分直到获得 t_0, 然后将其删除。如图12.3所示。



(a) 序列 $x_1, x_2, ..., x_5$ 对应森林 $[t_0, t_2]$

(b) 删除 x_5。直接删除 t_0。

(c) 删除 x_4。经过两次拆分后得到 $[t_0, t_0, t_1]$, 删除后得 $[t_0, t_1]$。

图 12.3: 从头部删除元素

$$\begin{aligned} extract\ ((x){:}ts) &= (x, ts) \\ extract\ ((s, t_1, t_2){:}ts) &= extract\ (t_1{:}t_2{:}ts) \end{aligned} \tag{12.5}$$

利用 $extract$ 即可实现对头部元素的删除:

$$\begin{cases} head &= fst \circ extract \\ tail &= snd \circ extract \end{cases} \tag{12.6}$$

其中 $fst\ (a, b) = a$, $snd\ (a, b) = b$ 分别返回一对值中的两个部分。

森林中的树实际上将元素划分为大小不同的区块。给定任意索引 $1 \leq i \leq n$, 我们先定位到对应的完全二叉树, 然后再进行一次树查找就可定位到元素。

1. 比较 i 和森林中第一棵树 t 的大小, 若 $i \leq size(t)$, 则元素在 t 中, 接下来在树 t 中进行查找;

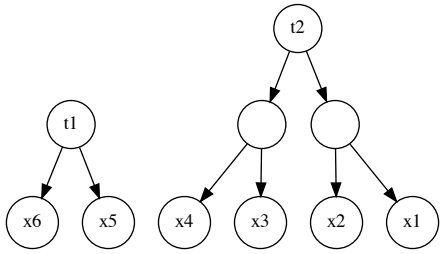
2. 否则, 令 $i' = i - size(t)$, 然后递归地在剩余的树中查找第 i' 个元素。

$$(t{:}ts)[i] = \begin{cases} i \leq size\ t: & lookup_T\ i\ t \\ \text{否则}: & ts[i - size\ t] \end{cases} \tag{12.7}$$

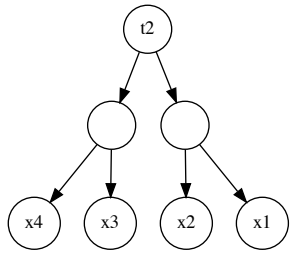
其中 $lookup_T$ 在树中进行二分查找。如果 $i = 1$, 我们返回根节点; 否则, 我们将树拆半, 然后递归查找:

$$\begin{aligned} lookup_T\ 1\ (x) &= x \\ lookup_T\ i\ (s, t_1, t_2) &= \begin{cases} i \leq \lfloor \frac{s}{2} \rfloor : & lookup_T\ i\ t_1 \\ \text{否则}: & lookup_T\ (i - \lfloor \frac{s}{2} \rfloor)\ t_2 \end{cases} \end{aligned} \tag{12.8}$$

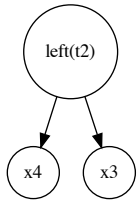
图12.4描述了在一个长度为 6 的序列中查找第 4 个元素的步骤。第一棵树大小为 $2 < 4$, 继续检查第二棵树, 并将索引更新为 $i' = 4 - 2$。接下来的树大小为 $4 > i' = 2$, 故待查找元素就在这棵树中。因为索引为 2, 不大于拆半的子树大小 $4/2 = 2$, 所以接下来检查左子树, 然后检查右侧的孙子分支, 最终得到要访问的元素。类似地, 我们也可以修改任意位置 i 的元素。



(a) $S[4], 4 > size(t_1) = 2$



(b) $S'[4 - 2] \Rightarrow lookup_T\ 2\ t_2$



(c) $2 \leq \lfloor \frac{size(t_2)}{2} \rfloor \Rightarrow lookup_T\ 2\ left(t_2)$

(d) $lookup_T\ 1\ right(left(t_2))$, 返回 x_3

图 12.4: 获取 $S[4]$

根据完全二叉树的性质, 对于含有 n 个元素的序列, 树木的棵数为 $O(\lg n)$。对于

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 $O(\lg n)$。

练习 12.1

如何处理索引越界情况?

12.3 数字表示

非负整数 n 的二进制形式和森林之间存在关系: $n = 2^0 e_0 + 2^1 e_1 + ... + 2^m e_m$, 其中 e_i 为第 i 位的值。若 $e_i = 1$, 则存在一棵大小为 2^i 的完全二叉树。向序列头部插入元素, 对应于二进制数加 1; 而删除对应二进制数减 1。我们称这种关系为数字表示[3]。为了明确表示这种对应, 我们为每个二进制位定义两个状态: 状态零 $Zero$ 表示不存在二叉树, 而状态一 $One\ t$ 表示存在二叉树 t。这样森林就可以表示为一组二进制状态的列表, 从而把插入实现为二进制数的增加。

$$\begin{aligned} add\ t\ [\,] &= [One\ t] \\ add\ t\ (Zero{:}ds) &= (One\ t) : ds \\ add\ t\ (One\ t'{:}ds) &= Zero : add\ (link\ t\ t')\ ds \end{aligned} \tag{12.9}$$

将树 t 插入森林对应二进制加法: 若森林为空, 我们创建状态 $One\ t$, 它是二进制数中的唯一位。相当于 $0+1=1$。若森林不空, 如果二进制首位数字是 $Zero$, 我们创建一个状态 $One\ t$ 替换掉 $Zero$。这相当于二进制加法 $(...digits...0)_2 + 1 = (...digits...1)_2$。例如 $6+1 = (110)_2+1 = (111)_2 = 7$。如果二进制首位是 $One\ t'$, 我们认为 t 和 t' 的大小相同。这是因为我们总是以一个叶子 $t_0 = (x)$ 开始插入, 待插入树的大小逐渐增长, 呈一个序列 $1, 2, 4, ..., 2^i, ...$。我们将 t 和 t' 链接起来, 递归地插入到剩余的数字中。而之前的 $One\ t'$ 被替换为 $Zero$。这相当于二进制加法 $(...digits...1)_2+1 = (...digits'...0)_2$。例如 $7+1 = (111)_2 + 1 = (1000)_2 = 8$。

接下来我们用二进制减法来表示删除。如果序列只含有一位 $One\ t$, 删除后序列变为空。这对应二进制减法 $1 - 1 = 0$。如果序列有多位, 并且首位是 $One\ t$, 我们将其替换为 $Zero$。这相当于二进制减法 $(...digits...1)_2 - 1 = (...digits...0)_2$。例如 $7-1 = (111)_2 - 1 = (110)_2 = 6$。如果首位是 $Zero$, 减法需要借位。我们递归地从剩余的数字中抽取树, 将其分拆成两棵树 t_1、t_2, 将 $Zero$ 替换成 $One\ t_2$, 并删除 t_1。这相当于二进制减法 $(...digits...0)_2 - 1 = (...digits'...1)_2$。例如 $4-1 = (100)_2 - 1 = (11)_2 = 3$。

$$\begin{aligned} minus\ [One\ t] &= (t, [\,]) \\ minus\ ((One\ t){:}ts) &= (t, Zero{:}ts) \\ minus\ (Zero{:}ts) &= (t_1, (One\ t_2){:}ts'), \text{其中} : (s, t_1, t_2) = minus\ ts \end{aligned} \tag{12.10}$$

数字表示并没有改变复杂度。我们使用聚合方法法, 分析在插入的分摊复杂度。考虑依次向空序列插入 $n = 2^m$ 个元素的过程。森林的二进制表示如表12.1:

i	二进制 (高... 低)
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
位变化次数	1, 1, 2, ... 2^{m-1}, 2^m

表 12.1: 插入 2^m 个元素的过程

二进制表示的最低位每次插入时都变化, 总共需要 2^m 次计算; 次低位每隔一次变化, 执行一次树的链接操作。共需要 2^{m-1} 次计算; 次高位总共只变化一次, 将所有的树链接成一棵大树。最后一个元素插入后, 最高位变为 1。将所有计算次数相加, 得到 $T = 1 + 1 + 2 + 4 + ... + 2^{m-1} + 2^m = 2^{m+1}$。平均每次插入操作的分摊复杂度为:

$$O(T/n) = O(\frac{2^{m+1}}{2^m}) = O(1) \tag{12.11}$$

这就证明了插入的分摊复杂度为常数时间。

练习 12.2

1. 实现数值表示序列的随机访问 $S[i], 1 \leq i \leq n$。其中 n 是序列长度。
2. 使用聚合法, 证明删除的分摊复杂度为常数时间。
3. 可以用长度为 2^m 的数组表示完全二叉树(m 是非负整数)。请用数组实现二叉树森林的插入和随机访问。并分析它们的分摊复杂度。

12.4 双数组序列

在上一章中, 我们给出过双数组队列。由于数组可以常数时间随机访问, 我们可以将其扩展为双数组序列。如图12.5, 按头对头的方式连接两个数组。在列表的头部插入时, 添加到 f 数组末尾; 向尾部插入时, 添加到 r 数组末尾。这样我们用一对数组表示列表 $S = (f, r)$, 令 $\text{FRONT}(S) = f$, $\text{REAR}(S) = r$。前后插入实现如下:

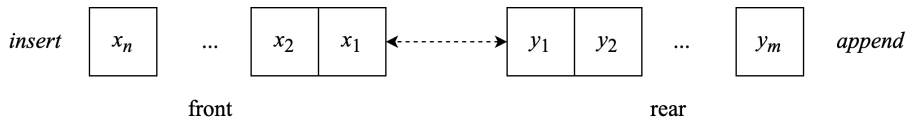


图 12.5: 双数组序列

```
1: function INSERT(x, S)
2:     APPEND(x, FRONT(S))
3: function APPEND(x, S)
4:     APPEND(x, REAR(S))
```

随机访问第 i 个元素时, 我们先判断 i 索引到 f 还是 r, 然后再定位到元素。若 $i \leq |f|$, 元素在 f 中。由于 f 和 r 头对头连接的, 所以 f 按照从右向左逆序索引元素。我们用 $|f| - i + 1$ 定位到元素; 如果 $i > |f|$, 元素在 r 中。元素是从左向右索引的, 我们用 $i - |f|$ 定位到元素。

```
1: function GET(i, S)
2:     f, r ← FRONT(S), REAR(S)
3:     n ← SIZE(f)
4:     if i ≤ n then
5:         return f[n − i + 1]                          ▷ 反向索引
6:     else
7:         return r[i − n]
```

删除可能把一个数组 f 或 r 变空, 而另一个仍有元素, 需要恢复平衡。当 f 或 r 等于 [ ] 时, 我们将另一数组分成两半, 然后将前一半反转形成一对新的数组。f、r 是对称的。我们也可以交换 f、r, 递归调用 BALANCE, 再把 f、r 交换回来。

```
1: function BALANCE(S)
2:     f ← FRONT(S), r ← REAR(S)
3:     n ← SIZE(f), m ← SIZE(r)
4:     if F = [ ] then
5:         k ← ⌊m/2⌋
6:         return (REVERSE(r[1...k]), r[(k + 1)...m])
7:     if R = [ ] then
8:         k ← ⌊n/2⌋
9:         return (f[(k + 1)...n], REVERSE(f[1...k]))
10:    return (f, r)
```

在每次删除时, 我们都检查 f、r 是否为空, 并触发平衡操作:

```
1: function REMOVE-HEAD(S)
2:     BALANCE(S)
3:     f, r ← FRONT(S), REAR(S)
4:     if f = [ ] then                                  ▷ S = ([], [x])
5:         r ← [ ]
6:     else
7:         REMOVE-LAST(f)
```

```
8: function REMOVE-TAIL(S)
9:     BALANCE(S)
10:    f, r ← FRONT(S), REAR(S)
11:    if r = [ ] then                                  ▷ S = ([x], [])
12:        f ← [ ]
13:    else
14:        REMOVE-LAST(r)
```

由于要进行反转, 双数组序列在最坏情况下性能为 $O(n)$, 其中 n 是元素个数。但是分摊复杂度是常数时间的。

练习 12.3

1. 证明双数组序列删除的分摊复杂度为常数时间。

12.5 可连接列表

虽然我们可以用 $O(\lg n)$ 时间在二叉树随机访问森林的头部进行插入、删除、索引, 但连接两个序列并不容易。我们不能简单地将所有二叉树合并到一起, 而需要不断链接大小相同的树。图12.6给出了一种可连接列表的实现。多叉树的根存储序列的第一个元素 x_1, 其它元素被分成若干片段保存在更小的序列中, 每个片段是一棵子树。这些子树由一个实时队列(见上一章)管理。我们把序列表示为 $(x_1, Q_x) = [x_1, x_2, ..., x_n]$。当需要连接另一个列表 $(y_1, Q_y) = [y_1, y_2, ..., y_m]$ 时, 我们将其入队到 Q_x 的尾部。连接运算定义如下。实时队列的入队性能为常数时间, 因此列表连接的性能也是常数时间的。

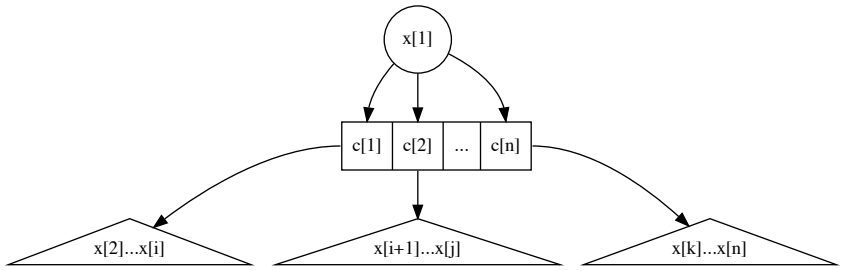
$$\begin{aligned} s \mathbin{+\!\!+} \varnothing &= s \\ \varnothing \mathbin{+\!\!+} s &= s \\ (x, Q) \mathbin{+\!\!+} s &= (x,\ push\ s\ Q) \end{aligned} \tag{12.12}$$

插入新元素 z 时, 我们先创建一个单元素列表 $(z, \varnothing)$, 然后将其连接起来。

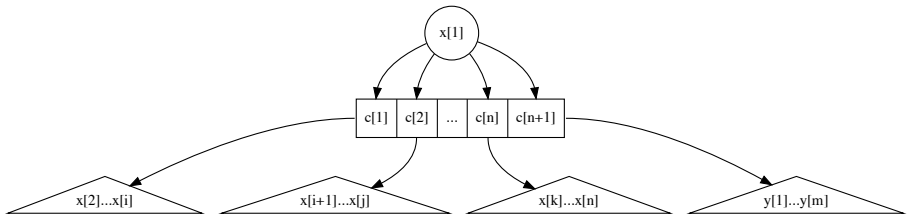
$$\begin{cases} insert\ x\ s &= (x, \varnothing) \mathbin{+\!\!+} s \\ append\ x\ s &= s \mathbin{+\!\!+} (x, \varnothing) \end{cases} \tag{12.13}$$

从可连接列表的头部删除元素需要单独的设计。x_1 为根节点, 删除后剩下的所有子树也都是由多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。

$$\begin{aligned} concat\ \varnothing &= \varnothing \\ concat\ Q &= (top\ Q) \mathbin{+\!\!+} concat\ (pop\ Q) \end{aligned} \tag{12.14}$$



(a) $(x_1, Q_x) = [x_1, x_2, ..., x_n]$



(b) 连接 $(y_1, Q_y) = [y_1, y_2, ..., y_m]$ 相当于入队 c_{n+1} 到 Q_x

图 12.6: 可连接列表

全部子树存储在实时队列中。我们将第一棵子树 c_1 出队，然后递归地将剩余的子树连接在一起成为 s，最后把 c_1、s 连接起来。我们使用 $concat$ 从头部删除元素。

$$tail\ (x, Q) = concat\ Q \tag{12.15}$$

算法 $concat$ 遍历了队列，逐步归并到一个最终的结果。这本质上相当于对 Q 进行叠加操作[10]。

$$\begin{aligned} fold\ f\ z\ \varnothing &= z \\ fold\ f\ z\ Q &= f\ (top\ Q)\ (fold\ f\ z\ (pop\ Q)) \end{aligned} \tag{12.16}$$

其中 f 是用于归并的二元函数，z 是零元。下面是一些队列叠加的例子，令 $Q = [1, 2, ..., 5]$。

$$\begin{aligned} fold\ (+)\ 0\ Q &= 1 + (2 + (3 + (4 + (5 + 0)))) = 15 \\ fold\ (\times)\ 1\ Q &= 1 \times (2 \times (3 \times (4 \times (5 \times 1)))) = 120 \\ fold\ (\times)\ 0\ Q &= 1 \times (2 \times (3 \times (4 \times (5 \times 0)))) = 0 \end{aligned}$$

我们可以利用叠加来定义 $concat$（克里化形式）：

$$concat = fold\ (+\!\!+)\ \varnothing \tag{12.17}$$

删除操作的性能在最坏情况下是线性的。当连续向空序列添加 n 个元素后，立即执行一次删除。此时多叉树中的 $n - 1$ 棵子树都是单元素的（只含有一个叶子节点）。$concat$ 需要 $O(n)$ 时间进行归并。如果插入、添加、删除、连接随机发生，则分摊复杂度是常数时间的。

练习 12.4

1. 证明可连接列表的删除操作的分摊复杂度为常数时间的。

12.6 手指树

二叉随机访问列表可以在头部用常数时间（分摊）插入、删除，以对数时间进行随机访问。但是难以向尾部添加元素、也无法进行快速连接。可连接列表能够用常数时间（分摊）进行连接，在头、尾部用常数时间插入。但不能用索引进行随机访问。这两个例子提示我们：1、需要某种方式快速访问头、尾以进行增删；2、带有递归的结构（例如树）可将随机访问转换成分而治之的搜索。手指树[66] 利用了这两点来实现序列[65]。树是否平衡对搜索性能至关重要。手指树利用了 2-3 树（一种 B-树）。一棵 2-3 树包含二或三棵子树，如：(t_1, t_2) 或 (t_1, t_2, t_3)。

```
data Node a = Br2 a a | Br3 a a a
```

我们定义一棵手指树为：

1. 或者为空 $\varnothing$；
2. 或者是单元素叶子 (x)；
3. 或者包含三部分：一棵子树和左、右手指，记为 (f, t, r)。每个手指是一个至多 3 个元素的列表[1]。

```
data Tree a = Empty
            | Lf a
            | Tr [a] (Tree (Node a)) [a]
```

12.6.1 插入

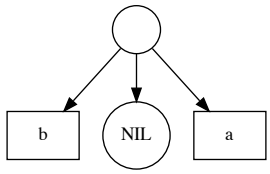
如图12.7和12.8所示。例 1 中 (a) 为 $\varnothing$，(b) 是插入一个元素后的结果。(c) 含有两个元素，分别在 f、r 手指中。如果继续插入元素，f 手指会超过 2-3 树的限制，如例 2(a) 所示。(b) 恢复平衡后，f 手指中有 2 个元素，中间部分是一个含有一棵 2-3 树的叶子。这些例子可以表示为：

$\varnothing$	`Empty`
(a)	`Lf a`
$([b], \varnothing, [a])$	`Tr [b] Empty [a]`
$([e, d, c, b], \varnothing, [a])$	`Tr [e, d, c, b] Empty [a]`
$([f, e], (d, c, b), [a])$	`Tr [f, e] Lf (Br3 d c b) [a]`

[1] 分别是英文前（front）、后（rear）的首字母。

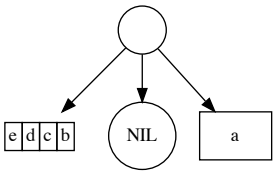
(a) $\varnothing$

(b) (x)

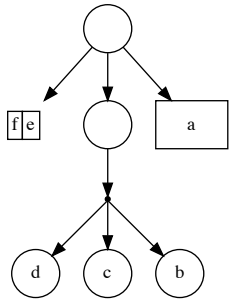


(c) $([b], \varnothing, [a])$

图 12.7: 手指树, 例 1



(a) 向 f 手指插入 3 个元素后, 超过了 2-3 树的限制, 不再平衡。



(b) 恢复平衡。f 手指含有 2 个元素; 中间部分为叶子, 包含一棵 2-3 树。

图 12.8: 手指树, 例 2

注意最后一个例子中, 中间部分的子树是一个叶子。手指树是递归的。除去 f、r 手指的中间部分是一棵更深的手指树, 定义为 $Tree\ (Node\ a)$。深度增加一级, 就多嵌套一级。上面的例子实际描述了向手指树插入元素的过程。我们可以归纳如下。向一棵手指树 T 中插入 a 时:

1. 如果 $T = \varnothing$, 则结果为单元素叶子 (a);
2. 如果 $T = (b)$ 是一个叶子, 结果为 $([a], \varnothing, [b])$;
3. $T = (f, t, r)$, 如果 f 中元素个数不超过 3, 将 a 插入到 f 中, 如果 f 中元素个数超过 3。将 f 中的后 3 个元素移入一棵新的 2-3 树 t', 递归地将 t' 插入到 t 中。最后将 a 插入到 f 中。

$$\begin{aligned} insert\ a\ \varnothing &= (x) \\ insert\ a\ (b) &= ([a], \varnothing, [b]) \\ insert\ a\ ([b, c, d, e], t, r) &= ([a, b], insert\ (c, d, e)\ t, r) \\ insert\ a\ (f, t, r) &= (a{:}f, t, r) \end{aligned} \tag{12.18}$$

除了递归插入外, 其它情况插入都需要常数时间。递归深度取决于树的高度 h, 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。递归可以分摊到其它情况中, 插入的分摊复杂度为常数时间[3] [65]。我们可以利用叠加连续将若干元素插入到树中:

$$xs \gg t = foldr\ insert\ t\ xs \tag{12.19}$$

练习 12.5

1. 消除递归, 用循环的方式实现手指树插入。

12.6.2 删除

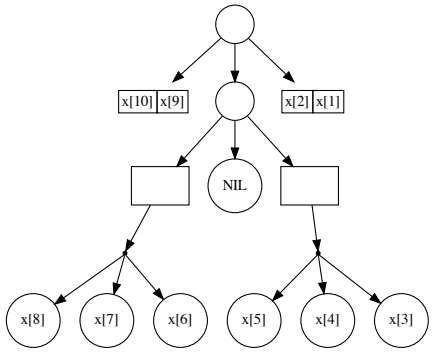
从头部删除可以看作对 $insert$ 进行逆操作。

$$\begin{aligned} extract\ (a) &= (a, \varnothing) \\ extract\ ([a], \varnothing, [b]) &= (a, (b)) \\ extract\ ([a], \varnothing, b{:}bs) &= (a, ([b], \varnothing, bs)) \\ extract\ ([a], t, r) &= (a, (toList\ f, t', r)), \text{其中} : (f, t') = extract\ t \\ extract\ (a{:}as, t, r) &= (a, (as, t, r)) \end{aligned} \tag{12.20}$$

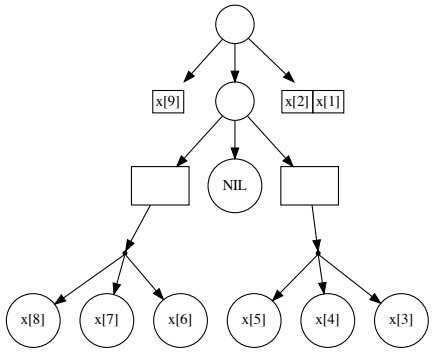
其中 $toList$ 将一棵 2-3 树转换为列表:

$$\begin{aligned} toList\ (a, b) &= [a, b] \\ toList\ (a, b, c) &= [a, b, c] \end{aligned} \tag{12.21}$$

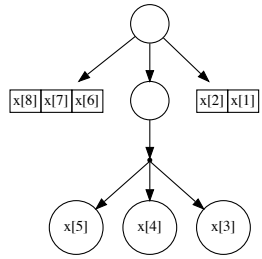
我们略过了错误情况(如从空树中删除)。如果手指树是单元素叶子, 结果为空树; 如果手指树只包含两个元素, 我们删除 f 中的元素, 结果为单元素的叶子; 如果 f 中只含有一个元素, 中间部分为空, 而 r 不空, 我们删除 f 中的唯一元素, 然后从 r 中“借”一个元素放入 f; 如果 f 只有一个元素, 而中间子树不空, 我们就递归地从子树中删除一个节点, 然后将这一节点中的内容转换成列表来代替 f。而原来 f 中的唯一元素被删除; 如果 f 包含一个以上的元素, 我们将第一个元素删除。图12.9展示了从序列头部删除两个元素的例子。



(a) 含有 10 个元素的树。



(b) 删除一个元素后, f 还剩一个元素。



(c) 再次删除一个元素, 从中间子树“借”一个节点, 将它从 2-3 树转换成列表, 作为新的 f。

图 12.9: 删除

使用 $extract$, 我们可以定义出 $head$ 和 $tail$:

$$\begin{cases} head & = fst \circ extract \\ tail & = snd \circ extract \end{cases} \tag{12.22}$$

练习 12.6

1. 消除递归, 用循环实现删除。

12.6.3 尾部操作

我们可以对称地实现尾部添加、删除。

$$\begin{array}{rcl} append\ \varnothing\ a & = & (a) \\ append\ (a)\ b & = & ([a], \varnothing, [b]) \\ append\ (f, t, [a, b, c, d])\ e & = & (f, append\ t\ (a, b, c), [d, e]) \\ append\ (f, t, r)\ a & = & (f, t, r +\!\!+ [a]) \end{array} \tag{12.23}$$

如果 r 中的元素不超过 4 个, 我们直接把新元素添加到 r 末尾。否则, 将 r 中的前三个元素取出, 构造一棵新的 2-3 树, 递归地添加到中间子树的尾部。类似地我们可以用左侧叠加连续将若干元素构造成一棵手指树:

$$t \ll xs = foldl\ append\ t\ xs \tag{12.24}$$

从尾部删除相当于添加的逆操作:

$$\begin{array}{rcl} remove\ (a) & = & (\varnothing, a) \\ remove\ ([a], \varnothing, [b]) & = & ((a), b) \\ remove\ (f, \varnothing, [a]) & = & ((initf, \varnothing, [lastf]), a) \\ remove\ (f, t, [a]) & = & ((f, t', toList\ r), a), \text{其中} : (t', r) = remove\ t \\ remove\ (f, t, r) & = & ((f, t, init\ r), last\ r) \end{array} \tag{12.25}$$

其中 $last$ 获取列表的最后一个元素, $init$ 返回其余部分(定义见第一章)。

12.6.4 连接

考虑两棵手指树都不为空的情况: $T_1 = (f_1, t_1, r_1)$、$T_2 = (f_2, t_2, r_2)$。我们用 f_1 作为连接结果中的 f, 用 r_2 作为结果中的 r。然后将 t_1、r_1、f_2、t_2 合并成中间子树。由于 r_1 和 f_2 都是节点的列表, 所以这等价于如下问题:

$$merge\ t_1\ (r_1 +\!\!+ f_2)\ t_2 = ?$$

t_1 和 t_2 也都是手指树，但它们比 T_1 和 T_2 深一级，若 T_1 中的元素类型为 a，则 t_1 中的元素类型为 $Node\ a$。我们递归地进行合并：保留 t_1 的 f 手指和 t_2 的 r 手指，然后将 t_1 和 t_2 的中间部分，t_1 的 r 手指和 t_2 的 f 手指合并。

$$\begin{aligned}
merge\ \varnothing\ ts\ t_2 &= ts \gg t_2 \\
merge\ t_1\ ts\ \varnothing &= t_1 \ll ts \\
merge\ (a)\ ts\ t_2 &= merge\ \varnothing\ (a{:}ts)\ t_2 \\
merge\ t_1\ ts\ (a) &= merge\ t_1\ (ts +\!\!+ [a])\ \varnothing \\
merge\ (f_1, t_1, r_1)\ ts\ (f_2, t_2, r_2) &= (f_1, merge\ t_1\ (nodes\ (r_1 +\!\!+ ts +\!\!+ f_2))\ t_2, r_2)
\end{aligned} \tag{12.26}$$

其中 $nodes$ 将若干元素组织成一组 2-3 树。这是因为中间子树中的元素类型，比手指中的元素类深一级。

$$\begin{aligned}
nodes\ [a, b] &= [(a, b)] \\
nodes\ [a, b, c] &= [(a, b, c)] \\
nodes\ [a, b, c, d] &= [(a, b), (c, d)] \\
nodes\ (a{:}b{:}c{:}ts) &= (a, b, c){:}nodes\ ts
\end{aligned} \tag{12.27}$$

这样我们可以用 $merge$ 来定义手指树的连接：

$$(f_1, t_1, r_1) +\!\!+ (f_2, t_2, r_2) = (f_1, merge\ t_1\ (r_1 +\!\!+ f_2)\ t_2, r_2) \tag{12.28}$$

比较这一定义和 (12.26)，连接操作本质上就是合并操作，我们可以给出下面更加一致的定义：

$$T_1 +\!\!+ T_2 = merge\ T_1\ [\,]\ T_2 \tag{12.29}$$

连接的性能取决于递归的合并操作。递归的深度为两棵树中较小的一棵。由于 2-3 树的平衡性，手指树的高度为 $O(\lg n)$ 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 $insert$ 8 次）为分摊常数时间，最坏情况为 $O(m)$，其中 m 是两棵树的高度差。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。

12.6.5 随机访问

我们的策略是把随机访问转换为树搜索。为了避免反复计算树的大小，我们给每个分枝节点增加一个 s 变量记录其包含的元素个数：(s, f, t, r)。

```
data Tree a = Empty
            | Lf a
            | Tr Int [a] (Tree (Node a)) [a]
```

$$\begin{aligned}
size\ \varnothing &= 0 \\
size\ (x) &= size\ x \\
size\ (s, f, t, r) &= s
\end{aligned} \tag{12.30}$$

这里 $size\ (x)$ 并不一定是 1。这是因为 x 可能是更深的节点，例如 $Node\ a$，而只有当第一层时大小才是 1。为此我们可以在插入或添加时，把每个元素 x 包装在一个单元里 $(x)_e$，规定这个单元的大小为 1，即：$size\ (x)_e = 1$（参见附录例子）。

$$\begin{cases} x \lhd t = insert\ (x)_e\ t \\ t \rhd x = append\ t\ (x)_e \end{cases} \tag{12.31}$$

以及：

$$\begin{cases} xs \ll t = foldr\ (\lhd)\ t\ xs \\ t \gg xs = foldl\ (\rhd)\ t\ xs \end{cases} \tag{12.32}$$

我们还需要获取 2-3 树的大小：

$$\begin{array}{rcl} size\ (t_1, t_2) & = & size\ t_1 + size\ t_2 \\ size\ (t_1, t_2, t_3) & = & size\ t_1 + size\ t_2 + size\ t_3 \end{array} \tag{12.33}$$

对于节点的列表（例如深一级的手指）我们可以用 $sum \circ (map\ size)$ 来计算大小。在插入和删除操作中，我们也需要更新树大小。增加大小信息后，给定一个位置 i，可以通过树搜索定位到相应的节点。手指树具有递归结构：(s, f, t, r)，我们令这些子结构的大小为：s_f、s_t、s_r，且 $s = s_f + s_t + s_r$。如果 $i \leq s_f$，则目标位于 f 中，我们接下来在 f 中查找；如果 $s_f < i \leq s_f + s_t$，则目标位于 t 中，我们递归在 t 中搜索；否则目标位于 r 中。除此之外，我们还需要处理叶子节点 (x) 的情况。我们用一对值 (i, t) 表示在数据结构 t 中 i 的位置，并定义查找操作 $lookup_T$ 如下：

$$\begin{array}{rcl} lookup_T\ i\ (x) & = & (i, x) \\ lookup_T\ i\ (s, f, t, r) & = & \begin{cases} i < s_f: & lookup_s\ i\ f \\ s_f \leq i < s_f + s_t: & lookup_N\ (lookup_T\ (i - s_f)\ t) \\ \text{否则}: & lookup_s\ (i - s_f - s_t)\ r \end{cases} \end{array} \tag{12.34}$$

这里：$s_f = sum\ (map\ size\ f), s_t = size\ t$，分别是手指树前两部分的大小。如果在叶子节点 (x) 中查找位于 i 的内容，结果为 (i, x)。否则我们判断 i 位于 (s, f, t, r) 中哪一部分。如果位于前后手指 f, r 中，我们依次查找手指列表中的每个元素。

$$lookup_s\ i\ (x{:}xs) = \begin{cases} i < size\ x: & (i, x) \\ \text{否则}: & lookup_s\ (i - size\ x)\ xs \end{cases} \tag{12.35}$$

如果 i 位于某个元素 x 中（$i < size\ x$），我们返回 (i, x)，否则我们继续查找后面的元素。如果 i 不位于前后手指 f, r，而在中间部分 t，我们递归地在更深层查找，得到

位置 (i', m)。这里 m 是一个 2-3 树，我们接下来在其中查找：

$$
\begin{aligned}
lookup_N\ i\ (t_1, t_2) &= \begin{cases} i < size\ t_1: & (i, t_1) \\ \text{否则}: & (i - size\ t_1, t_2) \end{cases} \\
lookup_N\ i\ (t_1, t_2, t_3) &= \begin{cases} i < size\ t_1: & (i, t_1) \\ size\ t_1 \le i < size\ t_1 + size\ t_2: & (i - size\ t_1, t_2) \\ \text{否则}: & (i - size\ t_1 - size\ t_2, t_3) \end{cases}
\end{aligned} \tag{12.36}
$$

我们此前为了计算大小把每个元素 x 都封装在 $(x)_e$ 中，最终我们需要从中把 x 取回：

$$
T[i] = \begin{cases} \text{若} lookup_T\ i\ T = (i', (x)_e): & Just\ x \\ \text{否则}: & Nothing \end{cases} \tag{12.37}
$$

我们利用了类型 $Maybe\ a = Nothing | Just\ a$ 来表示索引成功或没有找到[2]。随机访问需要递归在手指树中查找，递归次数取决于树的深度。由于手指树是平衡的，随机访问的复杂度为 $O(\lg n)$，其中 n 是存储的元素个数。

我们用手指树实现的序列在总体上有着均衡、良好的性能。头、尾操作的分摊复杂度为常数时间，可以在对数时间内进行连接、分割、随机索引[67]。到本章为止，我们介绍了最基本的数据结构。接下来可以使用它们解决一些典型问题。例如，我们可以用序列实现 MTF[3]编码算法[68]。MTF 把序列中任意位置 i 的元素移动到最前面：

$$
mtf\ i\ S = x \lhd S', \text{其中} (x, S') = extractAt\ i\ S
$$

在后面章节中，我们将介绍基本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；然后我们介绍字符串匹配算法和基本搜索算法。

练习 12.7

1. 在随机访问时，如何处理空树 $\varnothing$ 和索引越界的情况？
2. 实现 $cut\ i\ S$，在位置 i 把序列 S 分割开。

12.7 附录：例子程序

随机访问列表（森林）：

```
data Tree a = Leaf a
            | Node Int (Tree a) (Tree a)

type BRAList a = [Tree a]
```

[2]很多编程环境提供了类似的处理，例如 Java/C++ 中的 `Optional<T>` 类型

[3]英文 move to front 的缩写。它应用于 BWT (Burrows-Wheeler transform) 数据压缩算法。

```
size (Leaf _) = 1
size (Node sz _ _) = sz

link t1 t2 = Node (size t1 + size t2) t1 t2

insert x = insertTree (Leaf x) where
    insertTree t [] = [t]
    insertTree t (t':ts) = if size t < size t' then  t:t':ts
                           else insertTree (link t t') ts

extract ((Leaf x):ts) = (x, ts)
extract ((Node _ t1 t2):ts) = extract (t1:t2:ts)

head' = fst ∘ extract
tail' = snd ∘ extract

getAt i (t:ts) | i < size t = lookupTree i t
               | otherwise = getAt (i - size t) ts
  where
    lookupTree 0 (Leaf x) = x
    lookupTree i (Node sz t1 t2)
        | i < sz `div` 2 = lookupTree i t1
        | otherwise = lookupTree (i - sz `div` 2) t2
```

随机访问森林的数值表示:

```
data Digit a = Zero | One (Tree a)

type RAList a = [Digit a]

insert x = add (Leaf x) where
  add t [] = [One t]
  add t (Zero:ts) = One t : ts
  add t (One t' :ts) = Zero : add (link t t') ts

minus [One t] = (t, [])
minus (One t:ts) = (t, Zero:ts)
minus (Zero:ts) = (t1, One t2:ts') where
    (Node _ t1 t2, ts') = minus ts

head' ts = x where (Leaf x, _) = minus ts
tail' = snd ∘ minus
```

双数组序列:

```
Data Seq<K> {
    [K] front = [], rear = []
}

Int length(S<K> s) = length(s.front) + length(s.rear)

void insert(K x, Seq<K> s) = append(x, s.front)
```

```
void append(K x, Seq<K> s) = append(x, s.rear)

K get(Int i, Seq<K> s) {
    Int n = length(s.front)
    return if i < n then s.front[n - i - 1] else s.rear[i - n]
}
```

可连接列表:

```
data CList a = Empty | CList a (Queue (CList a))

wrap x = CList x emptyQ

x ++ Empty = x
Empty ++ y = y
(CList x q) ++ y = CList x (push q y)

fold f z q | isEmpty q = z
           | otherwise = (top q) `f` fold f z (pop q)

concat = fold (++) Empty

insert x xs = (wrap x) ++ xs
append xs x = xs ++ wrap x

head (CList x _) = x
tail (CList _ q) = concat q
```

手指树:

```
-- 2-3 树
data Node a = Tr2 Int a a
            | Tr3 Int a a a

-- 手指树
data Tree a = Empty
            | Lf a
            | Br Int [a] (Tree (Node a)) [a] -- size, front, mid, rear

newtype Elem a = Elem { getElem :: a } -- 封装元素

newtype Seq a = Seq (Tree (Elem a)) -- 序列

class Sized a where  -- 可计算大小
  size :: a → Int

instance Sized (Elem a) where
  size _ = 1  -- 元素的大小总为 1

instance Sized (Node a) where
  size (Tr2 s _ _) = s
  size (Tr3 s _ _ _) = s

instance Sized a ⇒ Sized (Tree a) where
```

```
  size Empty = 0
  size (Lf a) = size a
  size (Br s _ _ _) = s

instance Sized (Seq a) where
  size (Seq xs) = size xs

tr2 a b = Tr2 (size a + size b) a b
tr3 a b c = Tr3 (size a + size b + size c) a b c

nodesOf (Tr2 _ a b) = [a, b]
nodesOf (Tr3 _ a b c) = [a, b, c]

-- 左侧操作
x <| Seq xs = Seq (Elem x `cons` xs)

cons :: (Sized a) ⇒ a → Tree a → Tree a
cons a Empty = Lf a
cons a (Lf b) = Br (size a + size b) [a] Empty [b]
cons a (Br s [b, c, d, e] m r) = Br (s + size a) [a, b] ((tr3 c d e) `cons` m) r
cons a (Br s f m r) = Br (s + size a) (a:f) m r

head' (Seq xs) = getElem $ fst $ uncons xs
tail' (Seq xs) = Seq $ snd $ uncons xs

uncons :: (Sized a) ⇒ Tree a → (a, Tree a)
uncons (Lf a) = (a, Empty)
uncons (Br _ [a] Empty [b]) = (a, Lf b)
uncons (Br s [a] Empty (r:rs)) = (a, Br (s - size a) [r] Empty rs)
uncons (Br s [a] m r) = (a, Br (s - size a) (nodesOf f) m' r)
    where (f, m') = uncons m
uncons (Br s (a:f) m r) = (a, Br (s - size a) f m r)

-- 右侧操作
Seq xs |> x  = Seq (xs `snoc` Elem x)

snoc :: (Sized a) ⇒ Tree a → a → Tree a
snoc Empty a = Lf a
snoc (Lf a) b = Br (size a + size b) [a] Empty [b]
snoc (Br s f m [a, b, c, d]) e = Br (s + size e) f (m `snoc` (tr3 a b c)) [d, e]
snoc (Br s f m r) a = Br (s + size a) f m (r ++ [a])

last' (Seq xs) = getElem $ snd $ unsnoc xs
init' (Seq xs) = Seq $ fst $ unsnoc xs

unsnoc :: (Sized a) ⇒ Tree a → (Tree a, a)
unsnoc (Lf a) = (Empty, a)
unsnoc (Br _ [a] Empty [b]) = (Lf a, b)
unsnoc (Br s f@(_:_:_) Empty [a]) = (Br (s - size a) (init f) Empty [last f], a)
unsnoc (Br s f m [a]) = (Br (s - size a) f m' (nodesOf r), a)
    where (m', r) = unsnoc m
unsnoc (Br s f m r) = (Br (s - size a) f m (init r), a) where a = last r
```

```
-- 连接
Seq xs ++ Seq ys = Seq (xs >|< ys)

xs >|< ys = merge xs [] ys

t <<< xs = foldl snoc t xs
xs >>> t = foldr cons t xs

merge :: (Sized a) ⇒ Tree a → [a] → Tree a → Tree a
merge Empty es t2 = es >>> t2
merge t1 es Empty = t1 <<< es
merge (Lf a) es t2 = merge Empty (a:es) t2
merge t1 es (Lf a) = merge t1 (es++[a]) Empty
merge (Br s1 f1 m1 r1) es (Br s2 f2 m2 r2) =
    Br (s1 + s2 + (sum $ map size es)) f1 (merge m1 (trees (r1 ++ es ++ f2)) m2) r2

trees [a, b] = [tr2 a b]
trees [a, b, c] = [tr3 a b c]
trees [a, b, c, d] = [tr2 a b, tr2 c d]
trees (a:b:c:es) = (tr3 a b c):trees es

-- 索引
data Place a = Place Int a

getAt :: Seq a → Int → Maybe a
getAt (Seq xs) i | i < size xs = case lookupTree i xs of
                     Place _ (Elem x) → Just x
                 | otherwise = Nothing

lookupTree :: (Sized a) ⇒ Int → Tree a → Place a
lookupTree n (Lf a) = Place n a
lookupTree n (Br s f m r) | n < sf = lookups n f
                          | n < sm = case lookupTree (n - sf) m of
                                       Place n' xs → lookupNode n' xs
                          | n < s = lookups (n - sm) r
  where sf = sum $ map size f
        sm = sf + size m

lookupNode :: (Sized a) ⇒ Int → Node a → Place a
lookupNode n (Tr2 _ a b) | n < sa = Place n a
                         | otherwise = Place (n - sa) b
  where sa = size a
lookupNode n (Tr3 _ a b c) | n < sa = Place n a
                           | n < sab = Place (n - sa) b
                           | otherwise = Place (n - sab) c
  where sa = size a
        sab = sa + size b

lookups :: (Sized a) ⇒ Int → [a] → Place a
lookups n (x:xs) = if n < sx then Place n x
                   else lookups (n - sx) xs
  where sx = size x
```

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, 所以

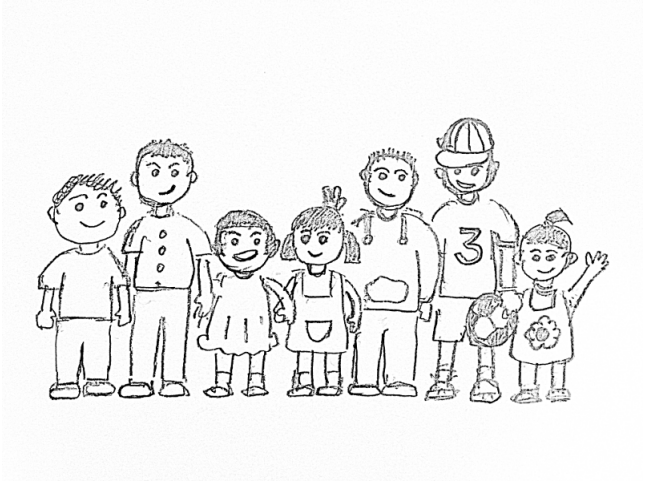


图 13.1: 安排小朋友们站成一队

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:

$$partition(p, L) = \begin{cases} (\phi, \phi) & : & L = \phi \\ (\{l_1\} \cup A, B) & : & p(l_1), (A, B) = partition(p, L') \\ (A, \{l_1\} \cup B) & : & \neg p(l_1) \end{cases} \tag{13.2}$$

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:

$$sort(L) = \begin{cases} \phi & : & L = \phi \\ sort(A) \cup \{l_1\} \cup sort(B) & : & L \neq \phi, (A, B) = partition(\lambda_x x \leq l_1, L') \end{cases} \tag{13.3}$$

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:

- 最左侧为 pivot, 当划分过程结束时, pivot 会被移动到最终的位置;

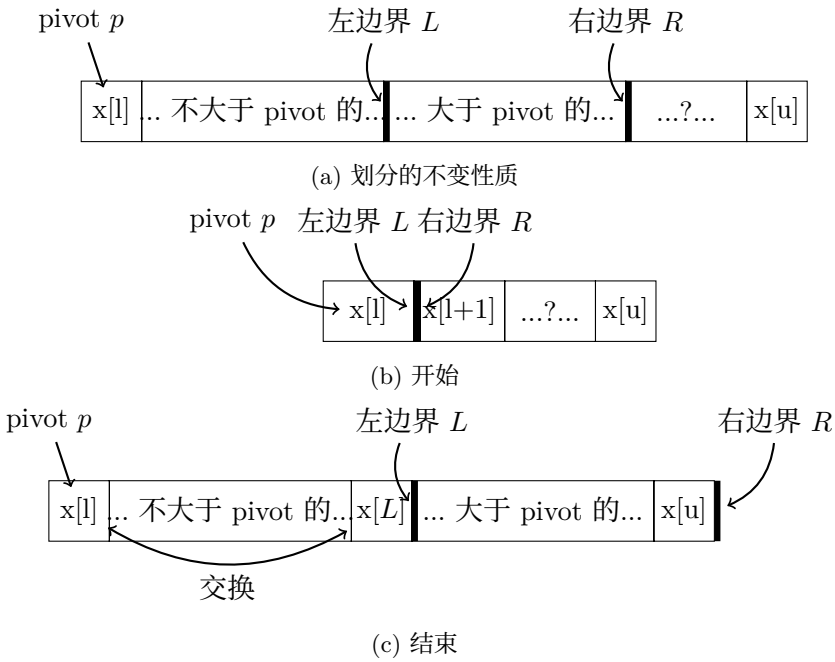


图 13.2: 使用最左边的元素作 pivot 划分一段数组

- 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。

3(l)	2(r)	5	4	0	1	6	7	开始, $pivot = 3$、$l = 1$、$r = 2$
3	2(l)(r)	5	4	0	1	6	7	$2 < 3$, 移动 $l (r = l)$
3	2(l)	5(r)	4	0	1	6	7	$5 > 3$, 继续
3	2(l)	5	4(r)	0	1	6	7	$4 > 3$, 继续
3	2(l)	5	4	0(r)	1	6	7	$0 < 3$
3	2	0(l)	4	5(r)	1	6	7	移动 l, 然后和 r 交换
3	2	0(l)	4	5	1(r)	6	7	$1 < 3$
3	2	0	1(l)	5	4(r)	6	7	移动 l, 然后和 r 交换
3	2	0	1(l)	5	4	6(r)	7	$6 > 3$, 继续
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)。

$$partition(p, L, A, B) = \begin{cases} (A, B) & : & L = \phi \\ partition(p, L', \{l_1\} \cup A, B) & : & p(l_1) \\ partition(p, L', A, \{l_1\} \cup B) & : & otherwise \end{cases} \tag{13.6}$$

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。

$$sort(L) = \begin{cases} \phi & : & L = \phi \\ sort(A) \cup \{l_1\} \cup sort(B) & : & otherwise \end{cases} \tag{13.7}$$

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。

$$sort'(L, S) = \begin{cases} S & : & L = \phi \\ ... & : & otherwise \end{cases}$$

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:

$$sort'(L, S) = \begin{cases} S & : & L = \phi \\ sort(A, \{l_1\} \cup sort(B, ?)) & : & otherwise \end{cases}$$

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。

$$sort'(L, S) = \begin{cases} S & : & L = \phi \\ sort(A, \{l_1\} \cup sort(B, S)) & : & otherwise \end{cases} \tag{13.8}$$

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。

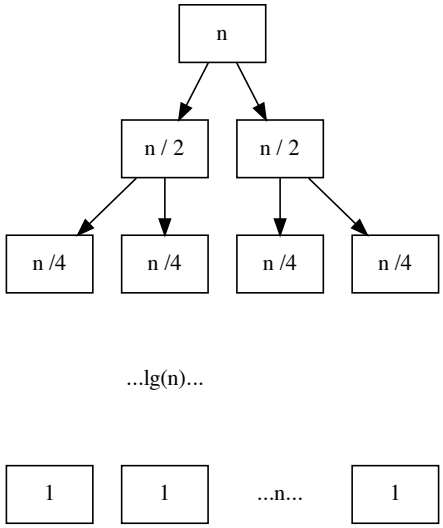


图 13.3: 最好情况下, 快速排序每次将序列划分成长度相等的两部分

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)

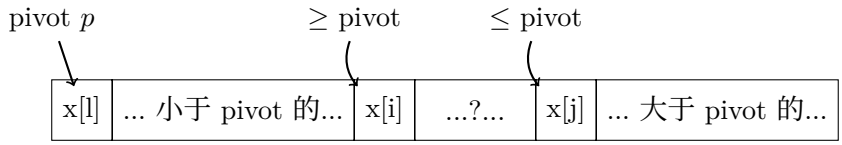
也可以用命令式的方法解决大量重复元素的问题。Robert Sedgewick 给出了一个划分方法[69]、[2], 使用两个指针, 一个从左向右移动, 另一个从右向左移动。开始的时候两个指针指向数组的左右边界。

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。

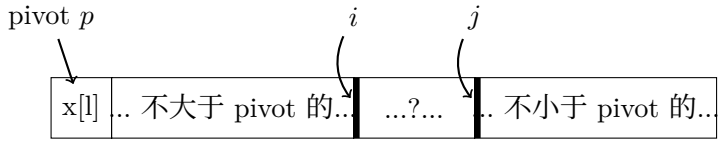
此时, 所有在左侧指针 i 之前的元素都严格小于 pivot, 而所有在右侧指针 j 之后的元素都严格大于 pivot。i 指向一个大于或等于 pivot 的元素; 而 j 指向一个小于或等于 pivot 的元素。图13.4 (a) 描述了此时的情形。

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);
    }
}
```

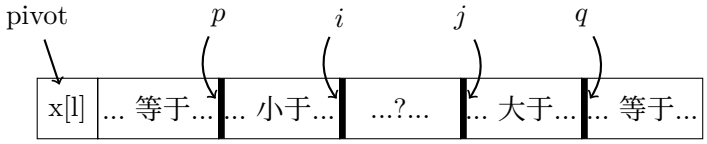
和此前介绍的 Lomuto 的划分算法相比, 可以发现这一算法的元素交换操作次数更少。这是因为它跳过了那些最终位置在 pivot 正确一侧的元素不进行交换。

三路划分

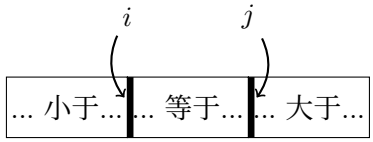
显然, 我们应该避免对重复元素进行不必要的交换操作。进一步, 可以利用“三分排序”(ternary sort, 也称作三路划分)的思路来改进算法, 所有严格小于 pivot 的元素

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。

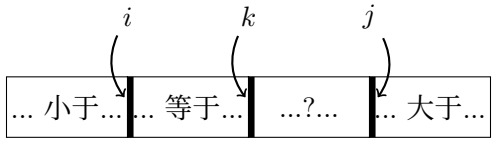


图 13.6: 基于 Lomuto 方法的三路划分

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

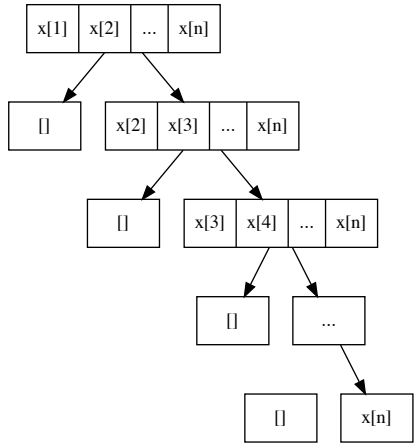
虽然三分快速排序(使用三路划分)能处理含有很多重复元素的序列, 但是仍然无法有效解决典型的最差情况。例如, 如果序列中的大部分元素已序时, 无论是升序还是降序, 划分的结果将会是两个长度不平衡的子序列, 一个包含少量的元素, 另一个包含剩余的部分。

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。

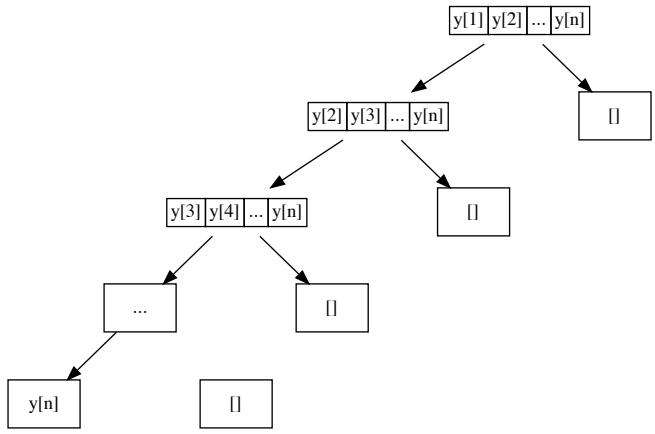
很容易给出更多的最差情况, 例如 $\{x_m, x_{m-1}, ..., x_2, x_1, x_{m+1}, x_{m+2}, ...x_n\}$, 其中 $\{x_1 < x_2 < ... < x_n\}$; 另一个例子是 $\{x_n, x_1, x_{n-1}, x_2, ...\}$。图13.8给出了它们的划分结果。

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

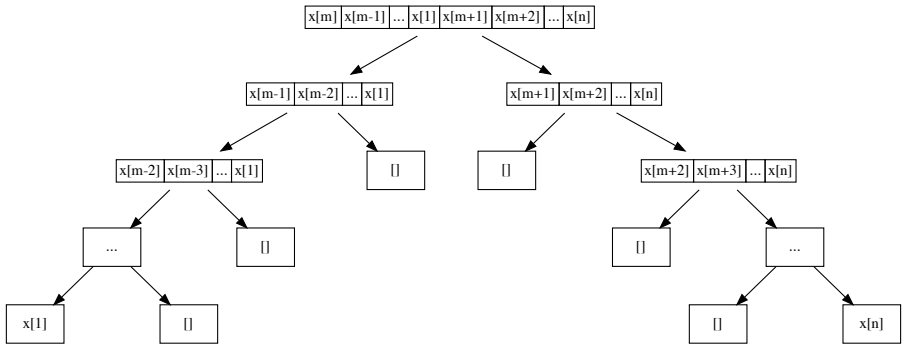


(a) 序列 $\{x_1 < x_2 < ... < x_n\}$ 的划分树, 每次划分时, 选择第一个元素为 pivot, 小于等于 pivot 的部分总为空。

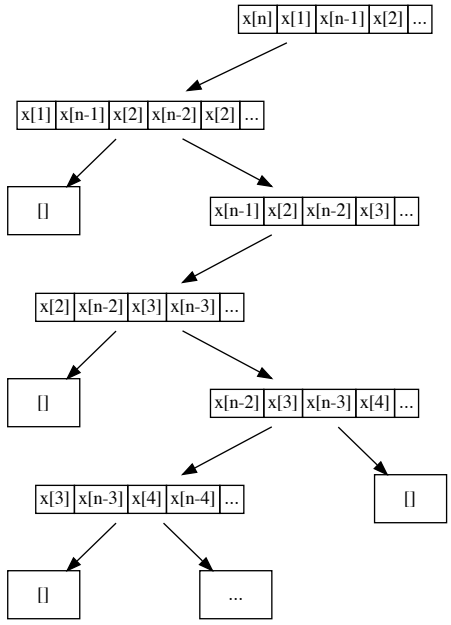


(b) 序列 $\{y_1 > y_2 > ... > y_n\}$ 的划分树, 每次划分时, 选择第一个元素为 pivot, 大于等于 pivot 的部分总为空。

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

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。

$$merge(A, B) = \begin{cases} A & : \quad B = \phi \\ B & : \quad A = \phi \\ \{a_1\} \cup merge(A', B) & : \quad a_1 \leq b_1 \\ \{b_1\} \cup merge(A, B') & : \quad otherwise \end{cases} \tag{13.31}$$

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

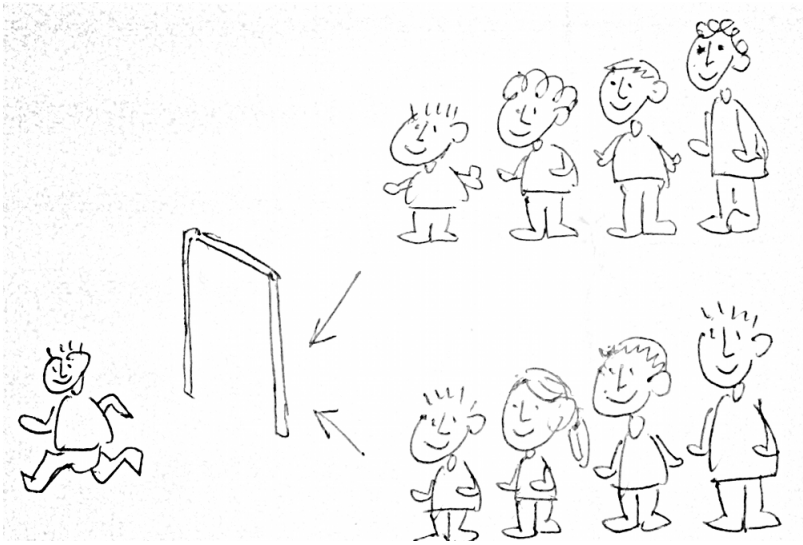


图 13.9: 两队孩子通过一扇门

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: $m \leftarrow \lfloor \frac{|A|}{2} \rfloor$
4: $X \leftarrow$ COPY-ARRAY($A[1...m]$)
5: $Y \leftarrow$ 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 ← 1, j ← 1, k ← 1
3:     m ← |X|, n ← |Y|
4:     while i ≤ m ∧ j ≤ n do
5:         if X[i] < Y[j] then
6:             A[k] ← X[i]
7:             i ← i + 1
8:         else
9:             A[k] ← Y[j]
10:            j ← j + 1
11:        k ← k + 1
12:    while i ≤ m do
13:        A[k] ← X[i]
14:        k ← k + 1
15:        i ← i + 1
16:    while j ≤ n do
17:        A[k] ← Y[j]
18:        k ← k + 1
19:        j ← 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。

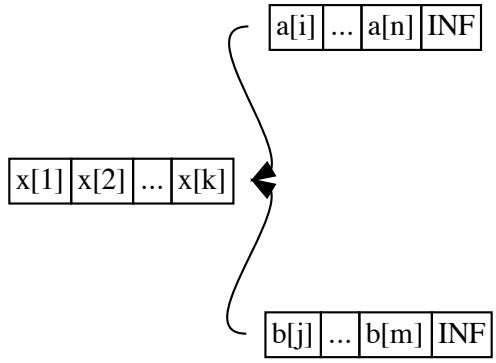


图 13.10: 使用 ∞ 作为 sentinels 来简化归并

```
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
        msort(xs, l, m);
```

[5]如果是按照单调非递增顺序排序, 则使用 $-\infty$

```
        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]
9:             j ← j + 1
```

```
10:         k ← k + 1
11:     while i ≤ m do
12:         B[k] ← A[i]
13:         k ← k + 1
14:         i ← i + 1
15:     while j ≤ u do
16:         B[k] ← A[j]
17:         k ← k + 1
18:         j ← j + 1
19:     for i ← from l to u do                          ▷ 复制回
20:         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:

$$split(L) = \begin{cases} (\phi, \phi) & : & L = \phi \\ (\{l_1\}, \phi) & : & |L| = 1 \\ (\{l_1\} \cup A, \{l_2\} \cup B) & : & otherwise, (A, B) = split(L'') \end{cases} \tag{13.33}$$

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 ≤ m ∧ m ≤ u do
3:         if A[l] < A[m] then
4:             l ← l + 1
5:         else
```

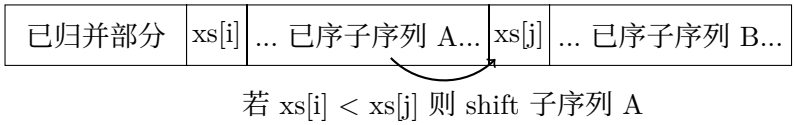


图 13.11: 死板原地归并

```
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。

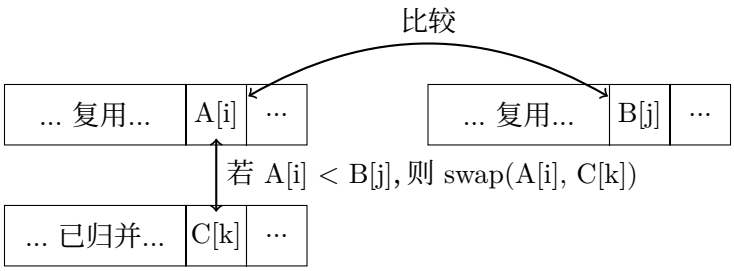


图 13.12: 归并时不覆盖工作区中的内容

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。

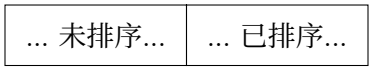


图 13.13: 数组中的一半被排序

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。

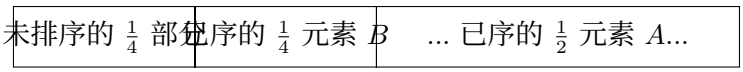


图 13.14: A 和 B 必须在某个时刻归并到一起

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]。

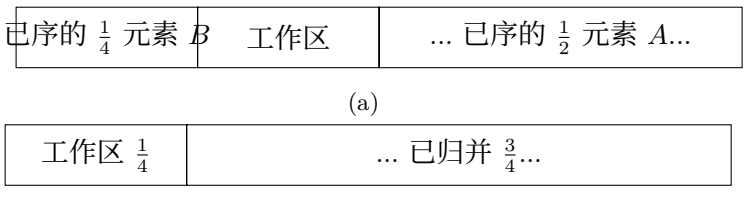


图 13.15: 利用工作区归并子数组 A 和 B

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 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: $(A, B) \leftarrow (\phi, \phi)$
3: **while** $L \neq \phi$ **do**
4: $p \leftarrow L$
5: $L \leftarrow \text{NEXT}(L)$
6: $A \leftarrow \text{LINK}(p, A)$
7: EXCHANGE $A \leftrightarrow B$
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。

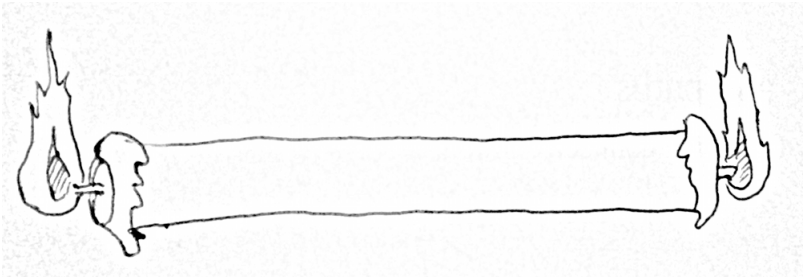


图 13.16: 从两端向中间燃烧的蜡烛

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

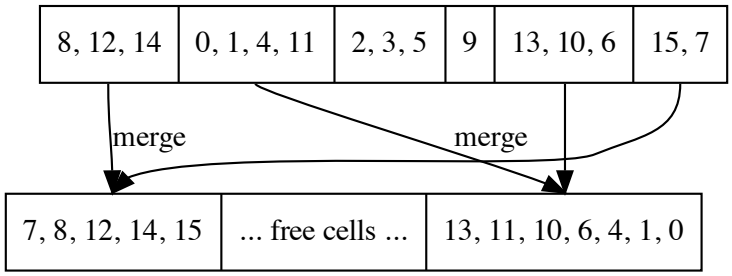


图 13.17: 自然归并排序

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。

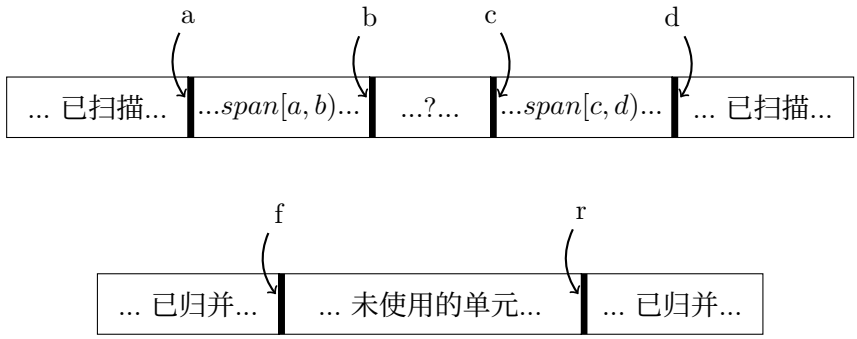


图 13.18: 自然归并排序时的不变性质

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)
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
4:              B[w] ← A[a]
5:              a ← a + 1
6:          else
7:              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]。

[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

```
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
```

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:

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。

```
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。

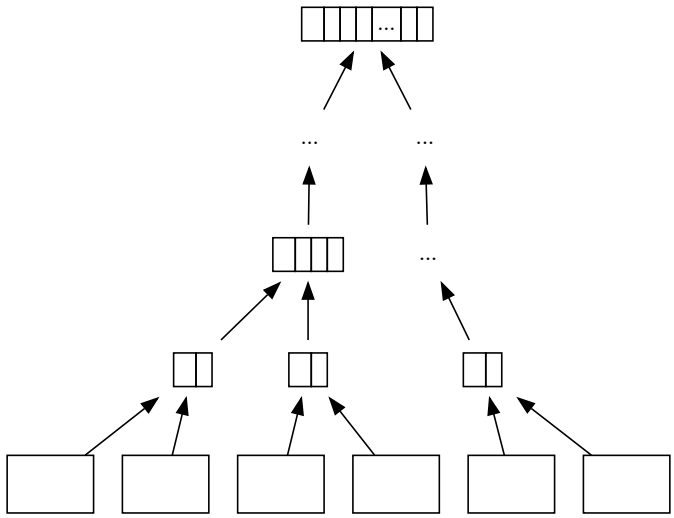


图 13.19: 自底向上归并排序

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。

```
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]
11:        N ← ⌈N/2⌉
12:    if B = φ then
13:        return φ
14:    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。

[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

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]。

[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？

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
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 $solve(a, y, X)$。

$$solve(a, y, X) = \begin{cases} x_1 & : & a^{x_1} = y \\ solve(a, y, X') & : & a^{x_1} < y \\ Err & : & otherwise \end{cases}$$

[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。

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。

$$\begin{bmatrix} 1 & 2 & 3 & 4 & ... \\ 2 & 4 & 5 & 6 & ... \\ 3 & 5 & 7 & 8 & ... \\ 4 & 6 & 8 & 9 & ... \\ ... & & & & \end{bmatrix}$$

图 14.1: 每行、每列都严格单调增的矩阵

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 $M_{i,j} = x$。

[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 $f(n)$。

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。

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。

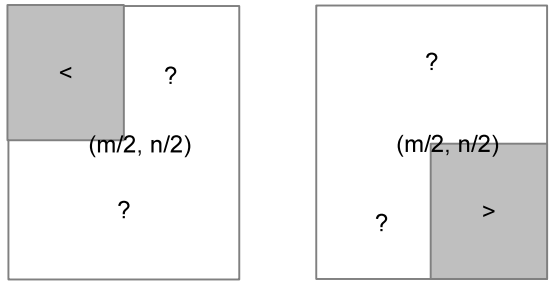


图 14.2: 左：中点的元素小于 x。所有灰色区域的元素都小于 x；右：中点的元素大于 x。所有灰色区域的元素都大于 x

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：

$$f(x,y)=\begin{cases} M_{x,y} & : \quad 1\leq x\leq m, 1\leq y\leq n \\ -1 & : \quad otherwise \end{cases}$$

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。

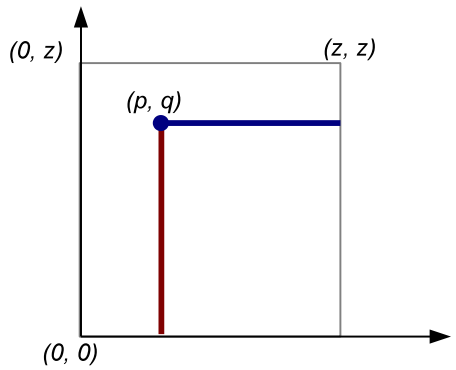


图 14.3: 从左上角搜索

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 $solve(f, z) = search(f, z, 0, z)$。

$$search(f, z, p, q) = \begin{cases} \phi & : \quad p > z \vee q < 0 \\ search(f, z, p + 1, q) & : \quad f(p, q) < z \\ search(f, z, p, q - 1) & : \quad f(p, q) > z \\ \{(p, q)\} \cup search(f, z, p + 1, q - 1) & : \quad otherwise \end{cases} \tag{14.9}$$

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 $2(z+1)$。

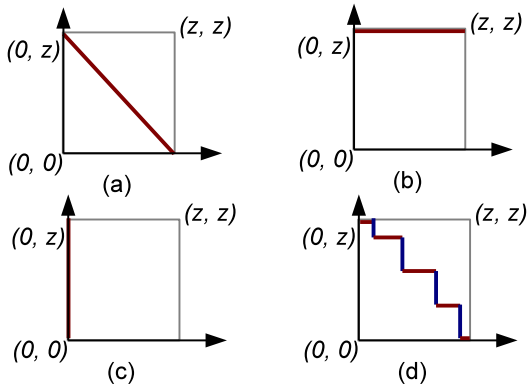


图 14.4: 最好和最差情况

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。

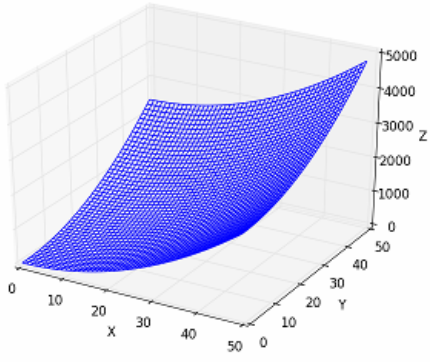


图 14.5: 函数 $f(x,y)=x^2+y^2$ 的图像

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。

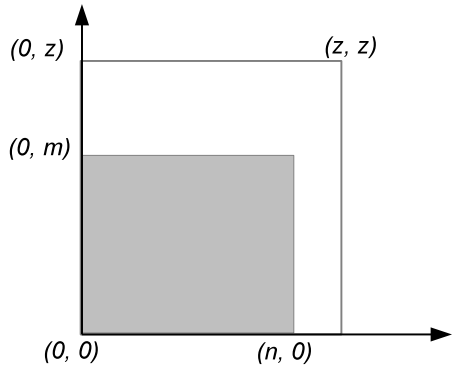


图 14.6: 缩小的灰色搜索区域

显然 m 和 n 可用穷举法找到：

$$\begin{aligned} m &= max(\{y | 0 \leq y \leq z, f(0, y) \leq z\}) \\ n &= max(\{x | 0 \leq x \leq z, f(x, 0) \leq z\}) \end{aligned} \tag{14.10}$$

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 $l \leq x \leq u$，使得 $f(x) = y$。而是要寻找 $l \leq x \leq u$ 使得满足不等式 $f(x) \leq y < f(x + 1)$。

$$bsearch(f, y, l, u) = \begin{cases} l & : & u \leq l \\ m & : & f(m) \leq y < f(m + 1), m = \lfloor \frac{l+u}{2} \rfloor \\ bsearch(f, y, m + 1, u) & : & f(m) \leq y \\ bsearch(f, y, l, m - 1) & : & otherwise \end{cases} \tag{14.11}$$

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 $solve(f, z) = search(f, z, 0, m)$:

$$search(f, z, p, q) = \begin{cases} \phi & : \quad p > n \vee q < 0 \\ search(f, z, p + 1, q) & : \quad f(p, q) < z \\ search(f, z, p, q - 1) & : \quad f(p, q) > z \\ \{(p, q)\} \cup search(f, z, p + 1, q - 1) & : \quad otherwise \end{cases} \tag{14.13}$$

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⌋
```

```
        if f(m) ≤ y then
            if y < f(m + 1) then
                return m
            l ← m + 1
        else
            u ← m
    return l
```

使用上述二分查找, 在开始 saddleback 搜索前, 先确定 m 和 n。

```
function SOLVE(f, z)
    m ← BINARY-SEARCH(λ_y · f(0, y), z, (0, z))
    n ← BINARY-SEARCH(λ_x · f(x, 0), z, (0, z))
    p ← 0, q ← m
    S ← φ
    while p ≤ n ∧ q ≥ 0 do
        z' ← f(p, q)
        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。

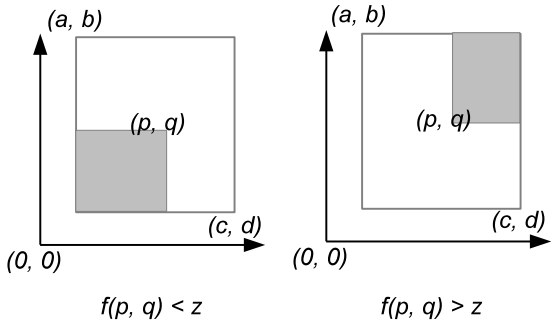
Saddleback 搜索的进一步改进

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。

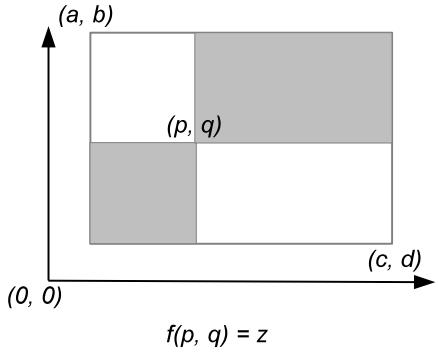
事实上, 我们忽略了另外的一个重要情况。我们观察矩形搜索区域中的任一一点, 如图14.7所示。

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。



(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

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。

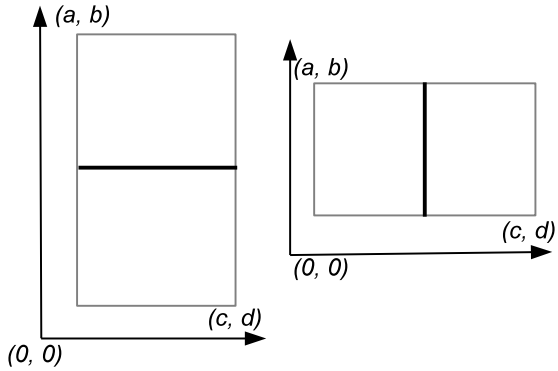


图 14.8: 沿较短的中线进行二分查找

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;
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)。

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)。

$$search_{(a,b),(c,d)} = \begin{cases} \phi & : & c < a \vee d < b \\ csearch & : & c - a < b - d \\ rsearch & : & otherwise \end{cases} \tag{14.14}$$

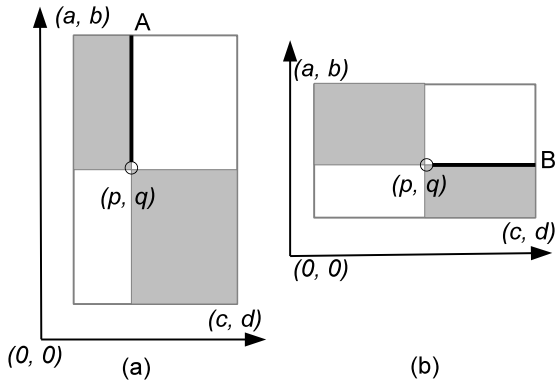


图 14.9: 递归搜索灰色的区域, 如果 $f(p,q) \neq z$, 还需要搜索加粗的线段

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。

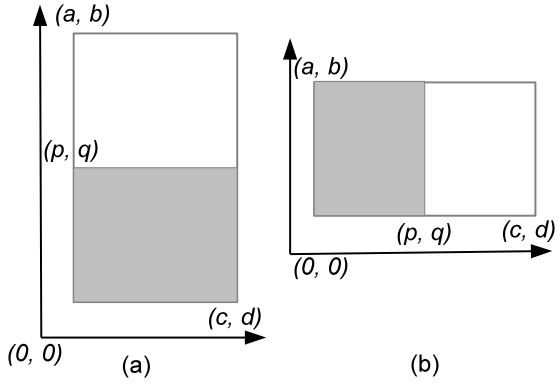


图 14.10: 沿中线进行二分查找时的特殊情况

$$csearch = \begin{cases} search_{(p,q-1),(c,d)} & : \quad z < f(p,q) \\ search_{(a,b),(p-1,q+1)} \cup \{(p,q)\} \cup search_{(p+1,q-1),(c,d)} & : \quad f(p,q) = z \\ search_{(a,b),(p,q+1)} \cup search_{(p+1,q-1),(c,d)} & : \quad otherwise \end{cases} \tag{14.15}$$

其中

$$\begin{aligned} q &= \lfloor \frac{b+d}{2} \rfloor) \\ p &= bsearch(\lambda_x \cdot f(x,q), z, (a,c)) \end{aligned}$$

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|
4:      loop
5:          i ← MAX-AT(A[l..k])
6:          j ← MIN-AT(A[k + 1..u])
7:          if A[i] < A[j] then
8:              break
9:          EXCHANGE A[l] ↔ A[j]
10:         EXCHANGE A[k + 1] ↔ A[i]
11:         l ← PARTITION(A, l, k)
12:         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;
}
```

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
```

[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]。

```
    if max > |A|50% then
        return m
    else
        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”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。

```
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
    return x
```

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

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)
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:     c ← 0
11:     for i ← 1 to |A| do
12:         if A[i] = x then
13:             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:

$$maj(c, n, L) = \begin{cases} c & : & L = \phi \\ maj(c, n+1, L') & : & l_1 = c \\ maj(l_1, 1, L') & : & n = 0 \wedge l_1 \neq c \\ maj(c, n-1, L') & : & otherwise \end{cases} \tag{14.19}$$

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)
```

[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。

```
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。

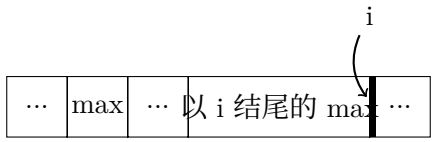


图 14.11: 扫描时的不变性质

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 $max_{sum}(0, 0, L)$。

$$max_{sum}(A, B, L) = \begin{cases} A & : & L = \phi \\ max_{sum}(A', B', L') & : & otherwise \end{cases} \tag{14.21}$$

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。

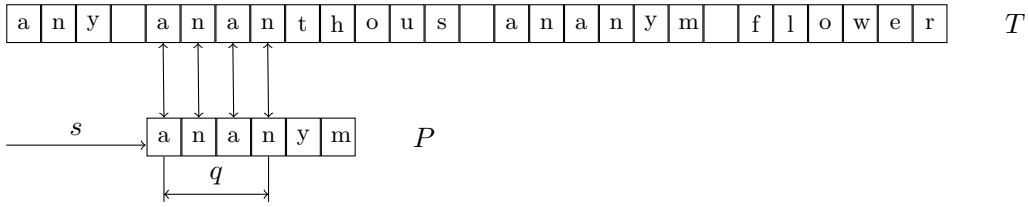
```
maxsum = msum 0 0 where
  msum a _ [] = a
  msum a b (x:xs) = let b' = max (x+b) 0
                        a' = max a b'
                    in msum a' b' xs
```

KMP

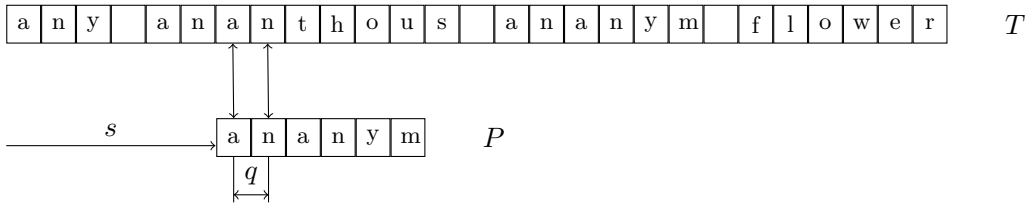
字符串搜索是一类很重要的问题。所有的文本编辑器软件都带有字符串搜索功能。在 Trie、Patricia 和后缀树章节，我们介绍了一些字符串搜索常用的数据结构。本

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。

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

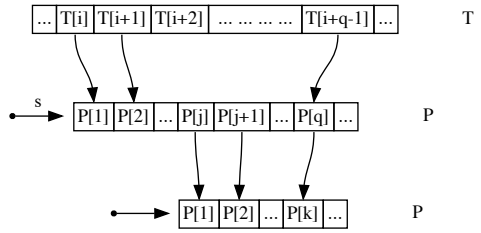


图 14.13: P_k 既是 P_q 的前缀, 也是 P_q 的后缀

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)
8:         if P[q + 1] = T[i] then
9:             q ← q + 1
10:        if q = m then
11:            found one solution at i − m
12:            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。

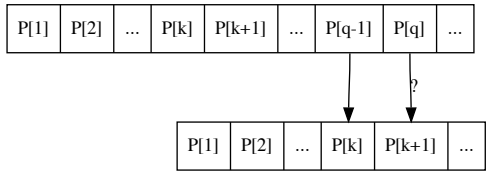


图 14.14: P_k 是 P_{q-1} 的后缀, 比较 $P[q]$ 和 $P[k+1]$

KMP 算法中, 构建前缀函数的过程可以描述如下:

```
1: function BUILD-PREFIX-FUNCTION(P)
2:     m ← |P|, k ← 0
3:     π(1) ← 0
4:     for q ← 2 to m do
5:         while k > 0 ∧ P[q] ≠ P[k + 1] do
6:             k ← π(k)
7:         if P[q] = P[k + 1] then
8:             k ← k + 1
9:         π(q) ← k
10:    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。

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

下面的 Python 例子程序实现了完整的 KMP 算法。

```
def kmp_match(w, p):
    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

[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。

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。

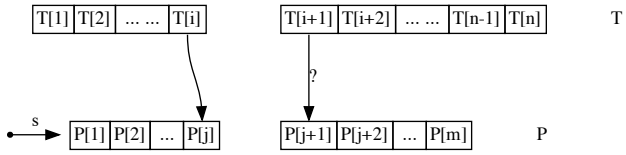


图 14.15: P 的前 j 个字符都相符, 接下来比较 $P[j+1]$ 和 $T[i+1]$

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:

$$prefixes(L) = \begin{cases} \{\phi\} & : & L = \phi \vee |L| = 1 \\ cons(\phi, map(\lambda_s \cdot cons(l_1, s), prefixes(L'))) & : & otherwise \end{cases} \tag{14.31}$$

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。

$$init(L) = \begin{cases} \phi & : & |L| = 1 \\ cons(l_1, init(L')) & : & otherwise \end{cases} \tag{14.35}$$

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:

$$\pi(P_p, P_s) = \begin{cases} (P_p, P_s) & : & P_p = \phi \\ fallback(init(P_p), cons(last(P_p), P_s)) & : & otherwise \end{cases} \tag{14.36}$$

其中

$$fallback(A, B) = \begin{cases} (A, B) & : & A \sqsupset P_p \\ (init(A), cons(last(A), B)) & : & otherwise \end{cases} \tag{14.37}$$

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。

$$match((P_p, P_s), L, R) = \begin{cases} True & : \quad P_s = \phi \\ False & : \quad otherwise \end{cases} \tag{14.39}$$

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

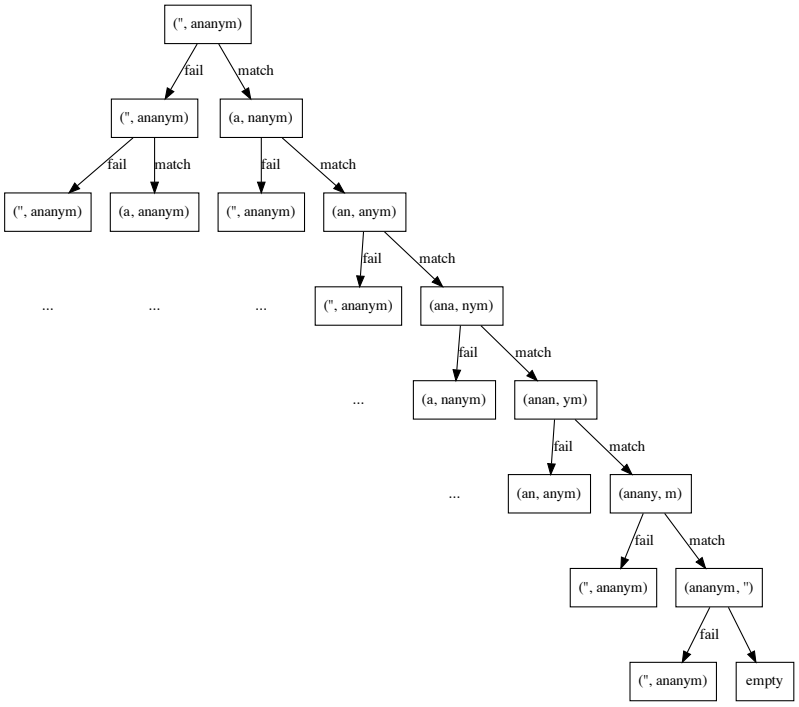


图 14.16: 从字符串“ananym”构造的无穷状态树

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:

$$\begin{aligned} F = & \quad \{0, 1, F_1 + F_0, F_2 + F_1, ...\} \\ = & \quad \{0, 1\} \cup \{x + y | x \in \{F_0, F_1, F_2, ...\}, y \in \{F_1, F_2, F_3, ...\}\} \\ = & \quad \{0, 1\} \cup \{x + y | x \in F, y \in F'\} \end{aligned} \tag{14.44}$$

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。

$$build(T, (P_p, P_s)) = \begin{cases} ((P_p, P_s), T, \phi) & : \quad P_s = \phi \\ ((P_p, P_s), T, build(trans(T, p), (P_p \cup \{p\}, P'_s))) & : \quad otherwise \end{cases} \tag{14.47}$$

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。

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。

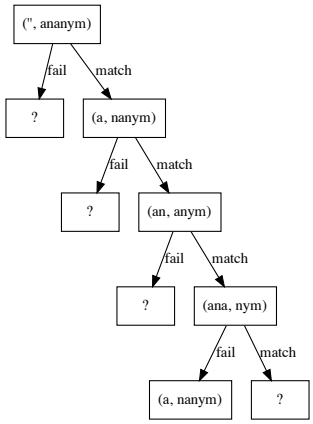


图 14.17: 在文本“anal”中搜索字符串“ananym”, 按需构造构造状态转移树

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。

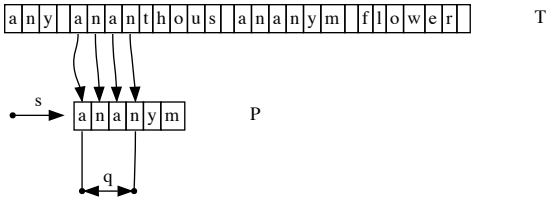
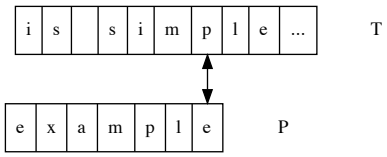


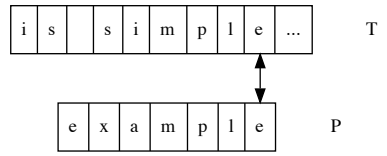
图 14.18: 因为字符 ‘h’ 没有出现在待搜索的字符串中, 向右侧平移的距离如果小于 6 都会匹配失败

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。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。

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



(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

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。

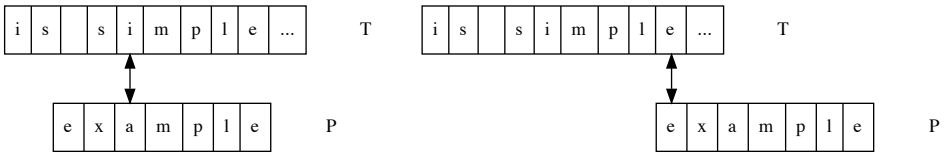


图 14.20: 即使字符 'i' 和 'a' 在中间位置匹配失败, 我们要使用字符 'e' 来查找平移的距离。得到结果 6(根据第一个 'e' 出现的位置计算, 需要跳过第二个 'e' 出现的位置以避免平移距离为 0)

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)
2:     for ∀c ∈ Σ do
3:         π[c] ← |P|
```

```
4:    for i ← 1 to |P| − 1 do                          ▷ 跳过最后一个位置
5:        π[P[i]] ← |P| − i
6:    s ← 0
7:    while s + |P| ≤ |T| do
8:        i ← |P|
9:        while i ≥ 1 ∧ P[i] = T[s + i] do              ▷ 从右侧开始扫描
10:           i ← i − 1
11:       if i < 1 then
12:           found one solution at s
13:           s ← s + 1                                ▷ 继续寻找下一个解
14:       else
15:           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。

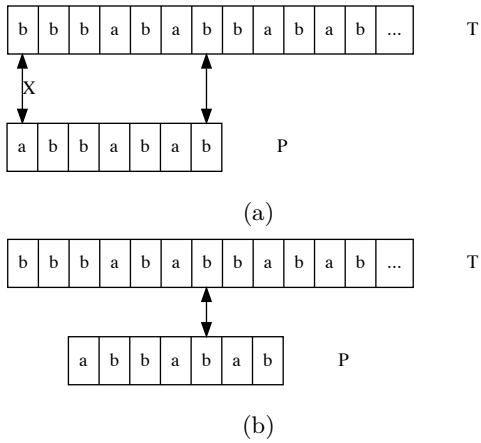


图 14.21: 根据不良字符规则，应向右平移 2 个单位，这样，下一个字符 ‘b’ 的位置相互对齐

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。

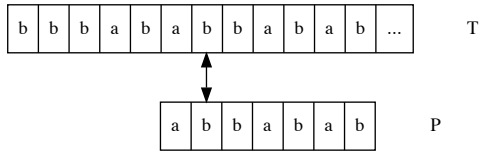


图 14.22: 由于前缀“ab”也是已匹配部分的后缀，我们可以向右平移使得“ab”对齐

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。

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

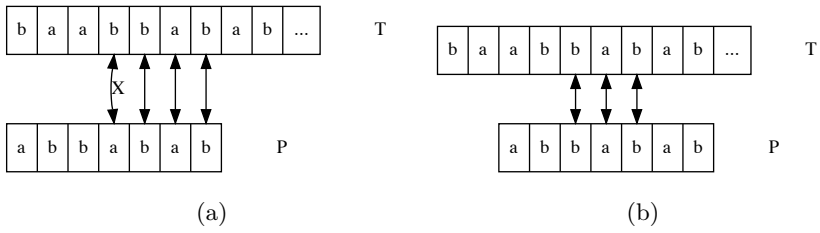


图 14.23: 已匹配的部分“bab”也在待搜索串的其它位置出现(从第 3 个字符到第 5 个字符)。我们只能向右平移 2 个单位以避免漏掉可能的解。

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。

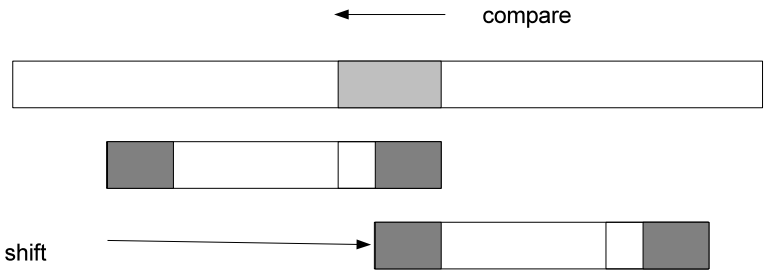
简单起见,记 P 中从第 i 个字符开始的后缀为 $\overline{P_i}$。即 $\overline{P_i}$ 为子串 $P[i]P[i+1]...P[m]$。

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。

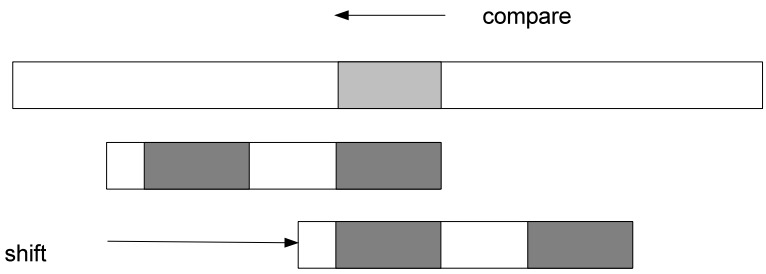
对于情况 2, 我们可以检查 P 的每个前缀, 包括 P_1, P_2, ..., P_{m-1}, 看它的最长后缀是否也是 P 的后缀。可以通过从左向右的另一轮扫描实现。

```
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 P̄_i ⊏ P then                            ▷ ⊏ 代表左侧是右侧的前缀
7:              l ← i
8:          π_s[i] ← l
9:      for i ← 1 to m do                              ▷ 第二轮循环处理情况 2
10:         s ← SUFFIX-LENGTH(P_i)
11:         if s ≠ 0 ∧ P[i − s] ≠ P[m − s] then
12:             π_s[m − s] ← m − i
13:     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。



(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

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。

1: **function** SUFFIX-LENGTH(P_i)
2: $m \leftarrow |P|$
3: $j \leftarrow 0$
4: **while** $P[m-j] = P[i-j] \wedge j < i$ **do**
5: $j \leftarrow j + 1$
6: **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: $s \leftarrow s + 1$ ▷ 继续寻找下一个解
13: **else**
14: $s \leftarrow s + max(\pi_b[T[s + m]], \pi_s[i])$

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])
```

```
        d ← MAX-SUM(A[m + 1...|A|])
        return MAX(a + b, c, d)

function MAX-FROM(A)
    sum ← 0, m ← 0
    for i ← 1 to |A| do
        sum ← sum + A[i]
        m ← MAX(m, sum)
    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,等高地图数据为

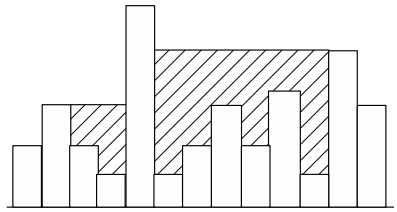


图 14.25: 灰色的区域表示积水

$\{0, 1, 0, 2, 1, 0, 1, 3, 2, 1, 2, 1\}$,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。

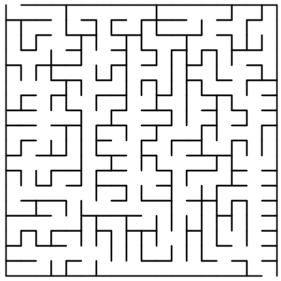


图 14.26: 一个迷宫的例子

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]。

$$solve(s, P) = \begin{cases} \{\{s\} \cup p | p \in P\} & : \quad s = e \\ concat(\{ \quad solve(s', \{\{s\} \cup p | p \in P\}) | & \\ \quad s' \in adj(s), \neg visited(s')\}) & : \quad otherwise \end{cases} \tag{14.52}$$

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。

$$adj((x, y)) = \{(x', y') | \quad (x', y') \in \{(x-1, y), (x+1, y), (x, y-1), (x, y+1)\}, \\ 1 \le x' \le M, 1 \le y' \le N, m_{x'y'} = 0\} \tag{14.54}$$

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]
    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],
```

[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。

```
      [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。

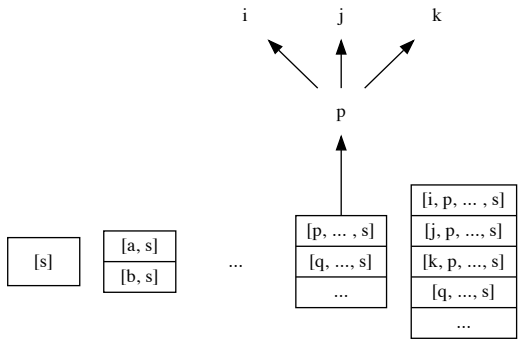


图 14.28: 栈初始时包含一个只有一个元素的列表。这一元素为起点 s。s 和点 a、b 连通。路径 $\{a, s\}$ 和 $\{b, s\}$ 被推回栈。在某一步, 以 p 结束的路径被弹出。p 和点 i、j 和 k 连通。这 3 个点被扩展为不同的选项, 并推回到栈中。除非所有的候选路径都失败, 否则不会尝试以 q 结尾的候选路径。

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 ← φ
3:      PUSH(S, {s})
4:      L ← φ                                          ▷ 结果列表
5:      while S ≠ φ do
6:          P ← POP(S)
7:          if e = p_n then
8:              ADD(L, P)
9:          else
10:             for ∀p ∈ ADJACENT(m, p_n) do
```

[10] $adjacent$ 函数的定义完全相同, 在此略过。

```
11:                 if p ∉ P then
12:                     PUSH(S, P ∪ {p})
13:     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
```

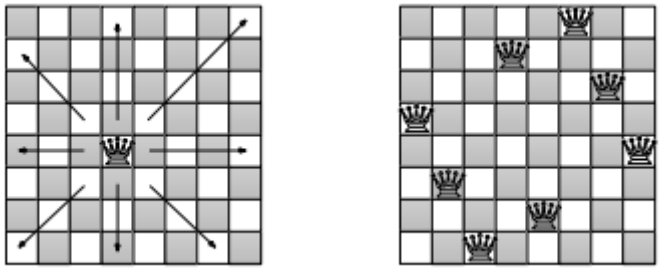
同样的例子迷宫可以用这一程序解决如下。

```
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。

$$solve(S, L) = \begin{cases} L & : & S = \phi \\ solve(S', \{s_1\} \cup L) & : & |s_1| = 8 \\ solve(\left\{ \begin{array}{ll} \{i\} \cup s_1 | & i \in [1, 8], \\ & i \notin s_1, \\ & safe(i, s_1) \end{array} \right\} \cup S', L) & : & otherwise \end{cases} \tag{14.59}$$

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: $S \leftarrow \{\phi\}$
3: $L \leftarrow \phi$ ▷ 保存所有解的列表
4: **while** $S \neq \phi$ **do**
5: $A \leftarrow$ POP(S) ▷ A 是某一中间布局

```
        if |A| = 8 then
            ADD(L, A)
        else
            for i ← 1 to 8 do
                if VALID(i, A) then
                    PUSH(S, A ∪ {i})
    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。

```
function VALID(x, A)
    y ← 1 + |A|
    for i ← 1 to |A| do
        if x = A[i] ∨ |y − i| = |x − A[i]| then
            return False
    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。

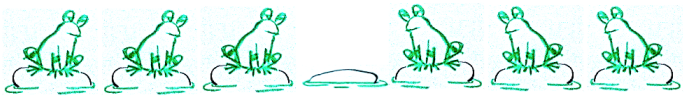


图 14.30: 跳跃的青蛙趣题

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。

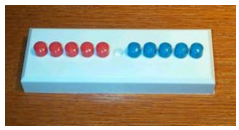
(b) 向右侧越过一只青蛙。

(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。

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。



(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

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 ← {{s}}
3:     M ← φ
4:     while S ≠ φ do
5:         s1 ← POP(S)
6:         if s1[1] = e then
7:             ADD(M, REVERSE(s1))
8:         else
9:             for ∀m ∈ MOVES(s1[1]) do
10:                PUSH(S, {m} ∪ s1)
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。

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

```
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。

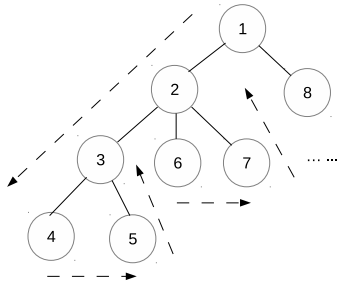


图 14.33: 深度优先搜索的顺序

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。

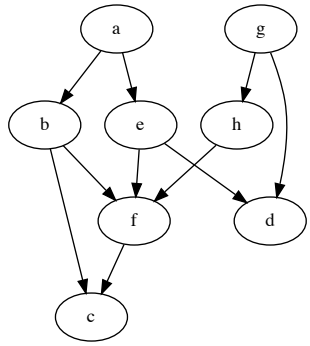


图 14.34: 一个有向图

```
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。

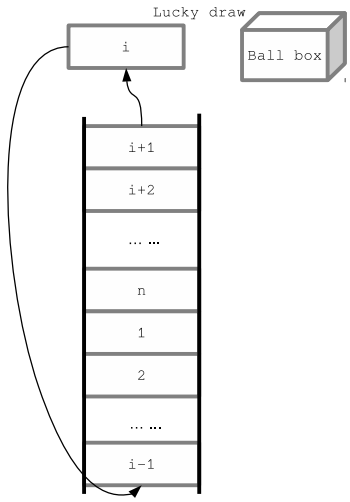


图 14.36: 抽奖游戏, 第 i 个人出队, 摸球。如果没有摸到黑球, 就站到队尾

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。

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

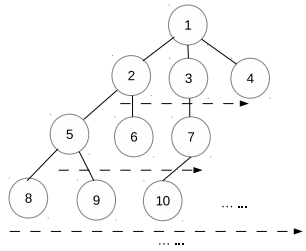


图 14.37: 从第一个状态开始, 检查第二步的所有选项 2、3、和 4; 然后检查第 3 层上的所有选项……

另外一个数字上。可行的移动方法定义如下:

$$mv(A,B)=\begin{cases}\{(A-8-i,B+8+i)|i\in\{0,1,2,4\},i=0\vee A\overline{\wedge}i\neq 0\} & : \quad B<8\\ \{(A+8+i,B-8-i)|i\in\{0,1,2,4\},i=0\vee B\overline{\wedge}i\neq 0\} & : \quad Otherwise\end{cases} \tag{14.64}$$

其中 $\overline{\wedge}$ 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。

$$solve(Q)=\begin{cases}\phi & : \quad Q=\phi\\ reverse(M) & : \quad A'=0\\ solve(EnQ'(Q',\left\{\begin{array}{ll}\{m\}\cup M| & m\in mv(m_1),\\ & valid(m,M)\end{array}\right\})) & : \quad otherwise\end{cases} \tag{14.65}$$

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

```
    S ← φ
    Q ← φ
    ENQ(Q, {(15, 0)})
    while Q ≠ φ do
        C ← DEQ(Q)
        if c_1 = (0, 15) then
            ADD(S, REVERSE(C))
        else
            for ∀m ∈ MOVES(C) do
                if VALID(m, C) then
                    ENQ(Q, {m} ∪ C)
    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。

```python
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]。

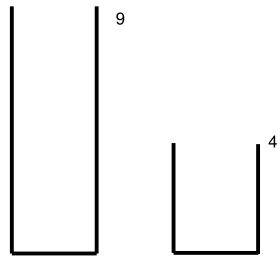


图 14.38: 两个瓶子的容积分别为 9 和 4

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。

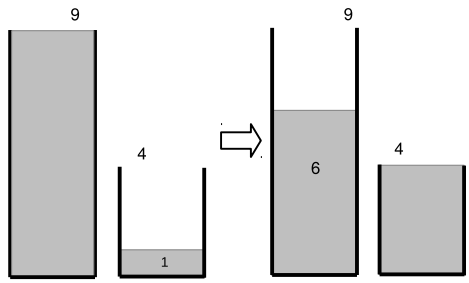


图 14.39: 最后两步

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。

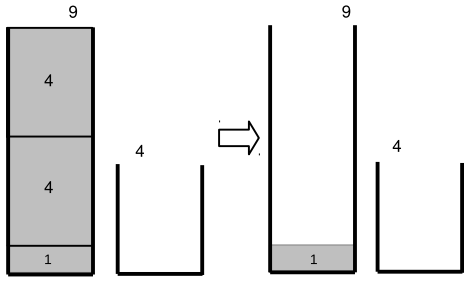


图 14.40: 将大瓶倒满, 然后倒入小瓶两次

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?

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 $a < b < 2a$。

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。

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

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:

$$solve'(Q) = \begin{cases} \phi & : & Q = \phi \\ reverse(S) & : & p = g \vee q = g \\ solve'(EnQ'(Q', \{\{s'\} \cup S' | s' \in try(S)\})) & : & otherwise \end{cases} \tag{14.76}$$

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。

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。

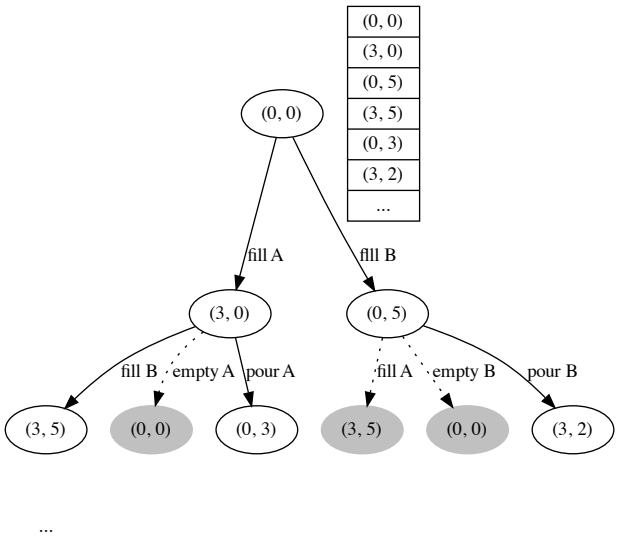


图 14.41: 所有尝试过的状态都存储于一个全局的列表中

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
```

```
        C ← EXPAND(s)
        for ∀c ∈ C do
            if c ≠ s ∧ ¬ VISITED(c) then
                PUSH-AND-RECORD(Q, c)
    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, %d\n", 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。

图 14.43: 日本的“箱子中的女儿”游戏

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。

$$M = \begin{bmatrix} 1 & 10 & 10 & 2 \\ 1 & 10 & 10 & 2 \\ 3 & 4 & 4 & 5 \\ 3 & 7 & 8 & 5 \\ 6 & 0 & 0 & 9 \end{bmatrix}$$

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。

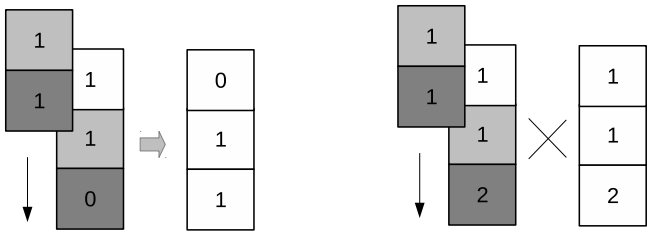


图 14.44: 左侧:两个标为 1 的格子都可以移动;右侧:上方标为 1 的格子虽然可以,但是下方标为 1 的格子和标为 2 的格子冲突。

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 $(\Delta x, \Delta y)$,其中 $|\Delta x| \leq 1$、$|\Delta y| \leq 1$。下面的等式,定义了移动是否可行:

$$
\begin{aligned}
&valid(L, k, \Delta x, \Delta y): \\
&\quad \forall (i,j) \in L[k] \Rightarrow \quad i' = i + \Delta y, j' = j + \Delta x, \\
&\qquad\qquad\qquad\qquad (1,1) \leq (i', j') \leq (5,4), M_{i'j'} \in \{k, 0\}
\end{aligned}
\tag{14.79}
$$

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)
```

[11] 也可以使用前面章节定义的手指树。

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])
```

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 SOLVE(L_0)
2:     H ← ||L_0||
3:     Q ← φ
4:     PUSH(Q, LINK(L_0, NIL))
5:     while Q ≠ φ do
6:         L ← POP(Q)
7:         if L[10] = {(4,2),(4,3),(5,2),(5,3)} then
8:             return L
9:         else
10:            for each L' ∈ EXPAND(L, H) do
11:                PUSH(Q, LINK(L', L))
12:                APPEND(H, ||L'||)
13:    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]
```

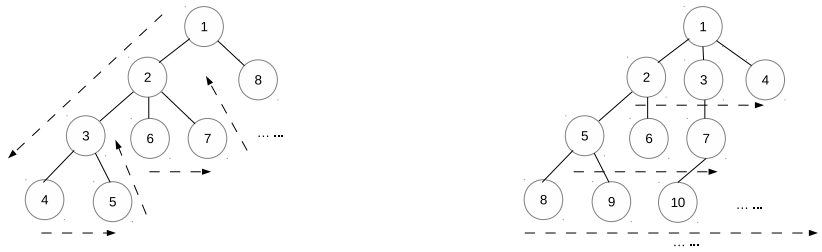
可以修改这一算法, 使得它不仅找出华容道的最优解, 还能找出所有的可能解法。这种情况下, 计算时间和搜索空间 V 成正比, 其中 V 包含从起始状态开始可以转换到的所有状态。若将所有这些状态存储在全局空间, 并使用父指针将后继状态链接起来, 则这一算法的空间复杂度也是 $O(V)$。

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。

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。

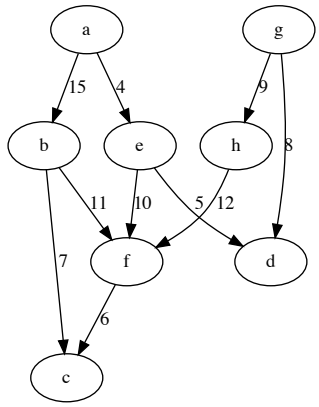


图 14.46: 带权重的有向图

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。

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:

文本“INTERNATIONAL”依照此码表被编码为 38 位的二进制数:

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

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

```
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。

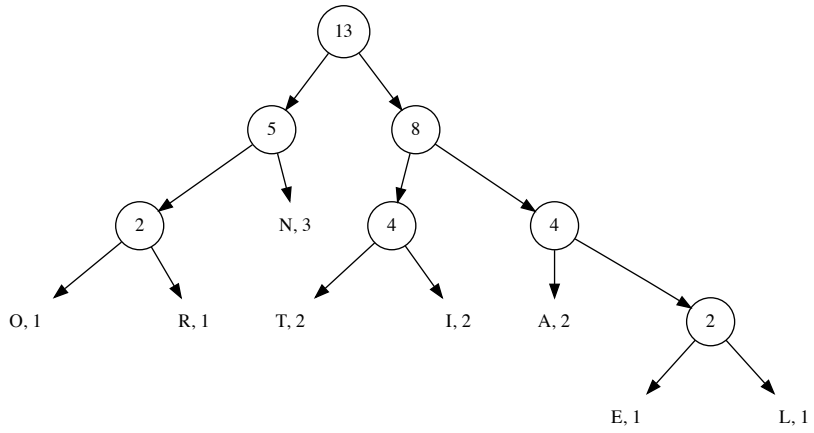


图 14.47: 一棵编码树

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。

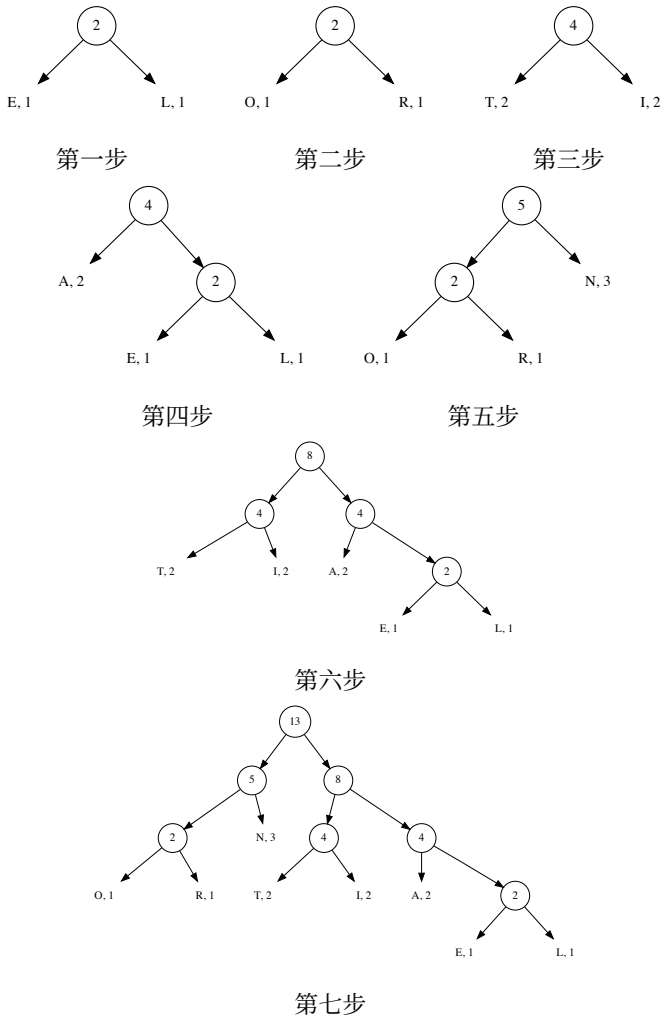


图 14.48: 构造一棵 Haffman 树的步骤

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。

$$reduce(T, H) = \begin{cases} T & : \quad H = \phi \\ build(insert(merge(T, top(H)), pop(H))) & : \quad otherwise \end{cases} \tag{14.90}$$

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。

$$extract''(Q, A) = \begin{cases} (T_b, (Q, A')) & : \quad Q = \phi \\ (T_a, (Q', A)) & : \quad A = \phi \vee T_a < T_b \\ (T_b, (Q, A')) & : \quad otherwise \end{cases} \tag{14.92}$$

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。

$$reduce'(T, (Q, A)) = \begin{cases} T & : \quad Q = \phi \wedge A = \phi \\ reduce'(extract''(push(Q'', merge(T, T')), A'')) & : \quad otherwise \end{cases} \tag{14.93}$$

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 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)
```

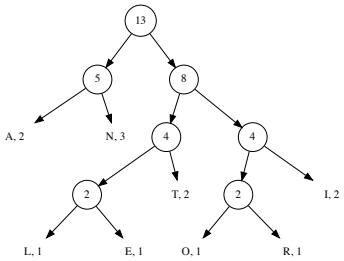
其中过程 DETACH(A) 将数组 A 的第一个元素取出返回, 并从数组中移除。在大多数命令式环境中, 从数组中移除第一个元素通常是一个较慢的线性时间操作。我们可以将树按照权重降序存储, 这样要移除的就是最后一个元素。速度为常数时间。下面的 C++ 例子程序实现了这一思路。

```
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);
    }
```

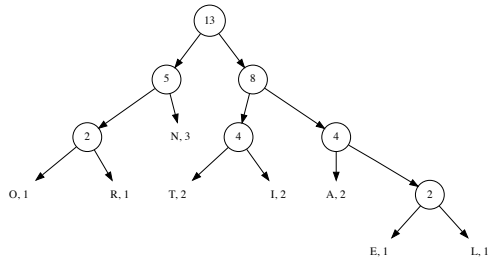
```
    return t;
}
```

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。

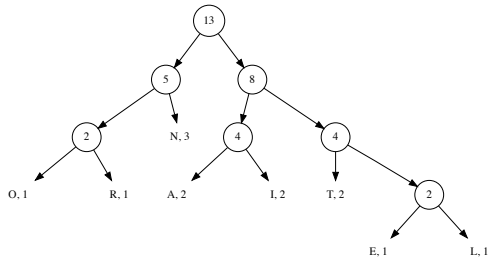
我们介绍了三种 Huffman 树的构造方法。虽然他们都符合 Huffman 提出的策略, 但是构造结果却不尽相同。图14.49给出了用三种不同方法构造的 Huffman 树。



(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
```

```
        C_m ← {c} ∪ CHANGE(X − c)
        if T[X] = φ ∨ |C_m| < |T[X]| then
            T[X] ← C_m
    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。

```
function FIBO(n)
    F = {1, 1, NIL, NIL, ...}
    for i ← 3 to n do
        F[i] ← F[i − 1] + F[i − 2]
    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。

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 $\{5, 1\}$;

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

- 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。

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)

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

```
    T ← {0, ∞, ∞, ...}
    S ← {NIL, NIL, ...}
    for i ← 1 to X do
        for c ∈ C, c ≤ i do
            if 1 + T[i − c] < T[i] then
                T[i] ← 1 + T[i − c]
                S[i] ← c
    while X > 0 do
        PRINT(S[X])
        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)。

[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。

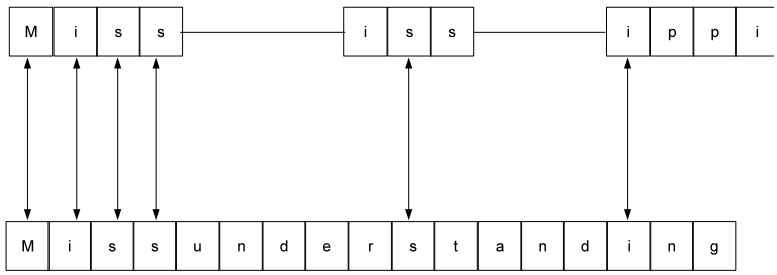


图 14.50: 最长公共子序列

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。

$$LCS(X, Y) = \begin{cases} \phi & : & X = \phi \vee Y = \phi \\ \{x_1\} \cup LCS(X', Y') & : & x_1 = y_1 \\ longer(LCS(X, Y'), LCS(X', Y)) & : & otherwise \end{cases} \tag{14.103}$$

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: $T \leftarrow$ 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。

```python
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。

$$solve(X, s) = \begin{cases} \phi & : & X = \phi \\ \{\{x_1\}\} \cup solve(X', s) & : & x_1 = s \\ solve(X', s) \cup \{\{x_1\} \cup S | S \in solve(X', s - x_1)\} & : & otherwise \end{cases} \tag{14.109}$$

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 ← Σ{x ∈ X, x < 0}
3:     s_u ← Σ{x ∈ X, x > 0}
4:     n ← |X|
5:     T ← {{False, False, ...}, {False, False, ...}, ...}        ▷ n × (s_u − s_l + 1)
6:     for i ← 1 to n do
7:         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
        if T[n − 1][s] then
            S ← S∪ GET(X, s, T, n − 1)
        if T[n − 1][s − X[n]] then
            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: $s_l \leftarrow \sum\{x \in X, x < 0\}$
3: $s_u \leftarrow \sum\{x \in X, x > 0\}$
4: $T \leftarrow \{\phi, \phi, ...\}$ ▷ $s_u - s_l + 1$
5: **for** $x \in X$ **do**
6: $T' \leftarrow$ DUPLICATE(T)
7: **for** $j \leftarrow s_l$ to s_u **do**
8: $j' \leftarrow j - x$
9: **if** $x = j$ **then**
10: $T'[j] \leftarrow T'[j] \cup \{x\}$
11: **if** $s_l \leq j' \leq s_u \wedge T[j'] \neq \phi$ **then**
12: $T'[j] \leftarrow T'[j] \cup \{\{x\} \cup S | S \in T[j']\}$
13: $T \leftarrow T'$
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
```

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。

[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。

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。

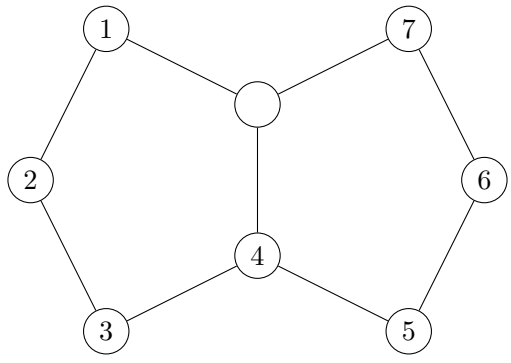


图 14.51: 康威滑动趣题

- 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。
- 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

和插入相比, 红黑树的命令式删除算法需要处理更多的情况。在普通二叉搜索树的删除算法之上, 我们通过旋转和重新染色恢复平衡性。删除黑色节点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”, 来保持黑色节点数目不变。下面的例子程序增加了双重黑色的定义:

```
data Color {RED, BLACK, DOUBLY_BLACK}
```

我们首先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, 我们通过双重黑色保持性质 5, 然后再做进一步修复。

```
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, RIGHT(x))                          ▷ 用右子树替换 x
7:      else if RIGHT(x) = NIL then
8:          q ← LEFT(x)
9:          REPLACE(x, LEFTx())                           ▷ 用左子树替换 x
10:     else
11:         y ← MIN(RIGHT(x))
12:         p ← PARENT(y)
13:         q ← RIGHT(y)
14:         KEY(x) ← KEY(y)
15:         copy data from y to x
16:         REPLACE(y, RIGHT(y))                          ▷ 用右子树替换 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
```

删除算法接受两个输入：根节点 T 和待删除节点 x。x 可以通过查找定位到。如果 x 含有为空的子树，我们将 x“切下”，并用另一子树 q 来替代 x。否则，我们在 x 的右子树中找到最小元素 y，用 y 替换 x。然后递归地将 y“切下”。如果 x 是黑色的，我们调用 MAKE-BLACK(p, q) 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 and q = NIL then
3:         return NIL                                  ▷ 树中只有一个叶子节点
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 p 和 q 都为空，我们在删除只有一个叶子节点的树，树变为空。如果父节点 p 不为空，而 q 为空，说明删除了一个黑色叶子节点。我们用 NIL 替换掉它。根据红黑树性质 3，NIL 是黑色的。我们把这一 NIL 变成“双重黑色”的 NIL 来保持性质 5 仍然成立。如果 p、q 都不为空，我们调用 BLACKEN(q) 检查 q 的颜色，如果是红色的，将它染成黑色，如果 q 已经是黑色的，将它染成双重黑色。接下来，我们通过旋转和重新染色，最终去掉“双重黑色”。这里有三种情况需要处理（[4]，292 页）。每种情况中，双重黑色的节点即可以是普通节点，也可以是双重黑色 NIL。

情况 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。我们可以通过旋转来修复。共有四种细分情况，它们全部可以变换到一种统一形式。如图A.1所示。

```
1: function DELETE-FIX(T, x, f)
2:     n ← NIL
3:     if f = True then                                ▷ x 是双重黑色 NIL
4:         n ← x
5:     if x = NIL then                                 ▷ 删除唯一的叶子
6:         return NIL
7:     while x ≠ T and COLOR(x) = B² do                ▷ x 是双重黑色，但不是根节点
8:         if SIBLING(x) ≠ NIL then                    ▷ 兄弟节点不为空
9:             s ← SIBLING(x)
10:            ...
11:            if s is black and LEFT(s) is red then
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)
```

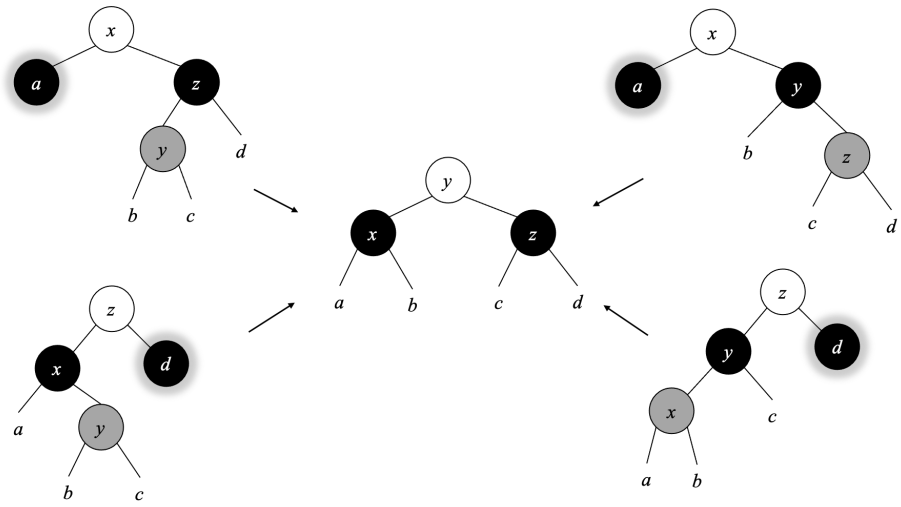


图 A.1: 双重黑色节点的兄弟为黑色，并且该兄弟节点有一个红色子节点。通过一次旋转操作修复。

```
15:                T ← RIGHT-ROTATE(T, PARENT(x))
16:            else                                          ▷ x 在右侧
17:                set x, PARENT(x), s, and LEFT(s) all black
18:                T ← LEFT-ROTATE(T, PARENT(x))
19:        else if s is black and RIGHT(s) is red then
20:            if x = LEFT(PARENT(x)) then                   ▷ x 在左侧
21:                set x, PARENT(x), s, and RIGHT(s) all black
22:                T ← RIGHT-ROTATE(T, PARENT(x))
23:            else                                          ▷ x 在右侧
24:                set x, PARENT(x), and RIGHT(s) all black
25:                T ← LEFT-ROTATE(T, s)
26:                T ← RIGHT-ROTATE(T, PARENT(x))
27:            ...
```

情况 2: 双重黑色节点的兄弟节点为红色。可以通过旋转，将双重黑色恢复为普通黑色。如图A.2所示，a 或 c 恢复为黑色。

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 是双重黑色 NIL
4:         n ← x
5:     if x = NIL then                                       ▷ 删除唯一的叶子
6:         return NIL
```

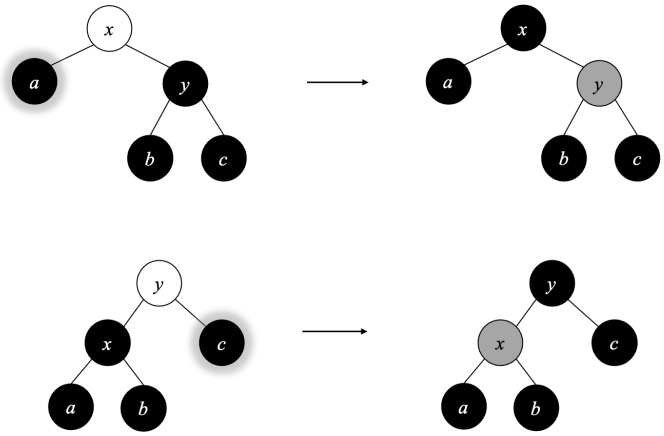


图 A.2: 双重黑色节点的兄弟节点为红色

```
7:    while x ≠ T and Color(x) = B² do
8:        if Sibling(x) ≠ NIL then
9:            s ← Sibling(x)
10:           if s is red then                          ▷ 兄弟节点为红色
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 and Left(s) is red then
18:               ...
```

情况 3: 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。可以将兄弟节点染成红色, 将双重黑色变回黑色, 然后将黑色向上传递。如图A.3所示, 有两种对称的情况。

上述三种情况中, 双重黑色节点的兄弟节点都不为空。否则, 我们直接将双重黑色恢复为普通黑色, 然后将黑色向上传递。如果最终到达根节点, 我们将根节点变为普通黑色, 并结束修复过程。另外, 如果在双重黑色在修复过程中被消除, 也可以终止。最后, 针对双重黑色 NIL 的情况, 我们将其恢复为普通 NIL。

```
1:  function Delete-Fix(T, x, f)
2:      n ← NIL
3:      if f = True then                                ▷ x 是双重黑色 NIL
4:          n ← x
5:      if x = NIL then                                 ▷ 删除唯一的叶子
6:          return NIL
```

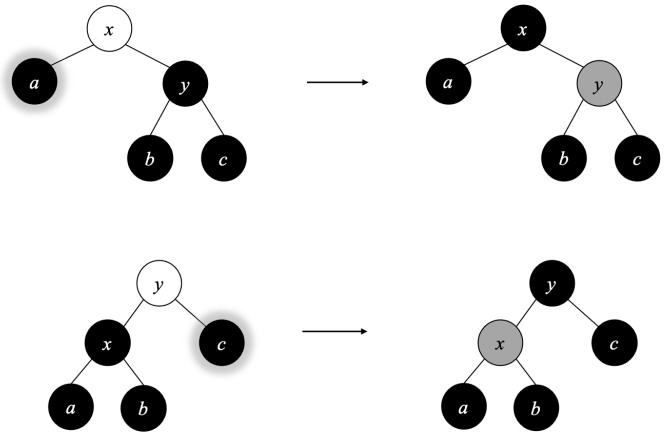


图 A.3: 向上传递黑色

```
7:  while x ≠ T and Color(x) = B² do
8:      if Sibling(x) ≠ NIL then                          ▷ 兄弟节点不为空
9:          s ← Sibling(x)
10:         if s is red then                              ▷ 兄弟节点为红色
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 and Left(s) is red then
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 and Right(s) is red then
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)
```

```
                T ← ROTATE-RIGHT(T, PARENT(x))
            else if s, LEFT(s), and RIGHT(s) are all black then
                set x black
                set s red
                BLACKEN(PARENT(x))
                x ← PARENT(x)
        else                                          ▷ 向上传递黑色
            set x black
            BLACKEN(PARENT(x))
            x ← PARENT(x)
    set T black
    if n ≠ NIL then
        replace n with NIL
    return T
```

修复时, 我们传入三个参数: 根节点 T、待修复节点 x(可能是双重黑色)、标记 f。如果 x 是双重黑色 NIL, 则 f 为真。此时我们用 n 来记录它, 并在最终修复完毕后, 用普通 NIL 替换 n。

下面的例子程序实现了红黑树删除算法。

```
Node del(Node t, Node x) {
    if x == null then return t
    var parent = x.parent;
    Node db = null;        //doubly black

    if x.left == null {
        db = x.right
        x.replaceWith(db)
    } else if x.right == null {
        db = x.left
        x.replaceWith(db)
    } else {
        var 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 == null);
    }
    remove(x)
    return t
}
```

其中 `makeBlack` 检查删除后节点是否变为双重黑色, 并处理双重黑色 NIL 的

特殊情况。

```
Node makeBlack(Node parent, Node x) {
    if parent == null and x == null then return null
    return if x == null
        then replace(parent, x, Node(0, Color.DOUBLY_BLACK))
        else blacken(x)
}
```

其中函数 replace(parent, x, y) 将 parent 的子节点 x,用 y 替换。

```
Node replace(Node parent, Node x, Node y) {
    if parent == null {
        if y ≠ null then y.parent = null
    } else if parent.left == x {
        parent.setLeft(y)
    } else {
        parent.setRight(y)
    }
    if x ≠ null then x.parent = null
    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 = if isRed(x) then Color.BLACK else Color.DOUBLY_BLACK
    return x
}
```

下面的例子程序实现了修复过程:

```
Node deleteFix(Node t, Node db, Bool isDBEmpty) {
    var dbEmpty = if isDBEmpty then db else null
    if db == null then return null    // delete the root
    while (db ≠ t and db.color == Color.DOUBLY_BLACK) {
        var s = db.sibling()
        var p = db.parent
        if (s ≠ null) {
            if isRed(s) {
                // the sibling is red
                p.color = Color.RED
                s.color = Color.BLACK
                t = if db == p.left then leftRotate(t, p)
                                    else rightRotate(t, p)
            } else if isBlack(s) and isRed(s.left) {
                // the sibling is black, and one sub-tree is red
                if db == p.left {
                    db.color = Color.BLACK
                    p.color  = Color.BLACK
                    s.left.color = p.color
                    t = rightRotate(t, s)
                    t = leftRotate(t, p)
                } else {
```

```
                db.color = Color.BLACK
                p.color  = Color.BLACK
                s.color = p.color
                s.left.color = Color.BLACK
                t = rightRotate(t, p)
            }
        } else if isBlack(s) and isRed(s.right) {
            if (db == p.left) {
                db.color = Color.BLACK
                p.color  = Color.BLACK
                s.color  = p.color
                s.right.color = Color.BLACK
                t = leftRotate(t, p)
            } else {
                db.color = Color.BLACK
                p.color  = Color.BLACK
                s.right.color = p.color
                t = leftRotate(t, s)
                t = rightRotate(t, p)
            }
        } else if isBlack(s) and isBlack(s.left) and
                  isBlack(s.right) {
            // the sibling and both sub-trees are black.
            // move blackness up
            db.color = Color.BLACK
            s.color  = Color.RED
            blacken(p)
            db = p
        }
    } else { // no sibling, move blackness up
        db.color = Color.BLACK
        blacken(p)
        db = p
    }
}
t.color = Color.BLACK
if (dbEmpty ≠ null) { // change the doubly black nil to nil
    dbEmpty.replaceWith(null)
    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, NIL 也是黑色的。

```
Bool isBlack(Node x) = (x == null or x.color == Color.BLACK)

Bool isRed(Node x) = (x ≠ null and x.color == Color.RED)
```

算法在结束前修复双重黑色 NIL, 调用 `Node` 中的 `replaceWith` 进行替换。

```
data Node<T> {
```

```
    //...
    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, 或者在双重黑色消失时终止。对于含有 n 个节点的红黑树, 其复杂度为 $O(\lg n)$。

练习 A.1

1. 编写程序判断一棵树是否满足红黑树的 5 条性质, 并用来验证红黑树的删除算法。

附录 B　AVL 树——证明和删除算法

B.1　插入后的高度变化

向 AVL 树插入元素后, 高度变化存在四种情况:

$$\begin{aligned}\Delta H &= |T'| - |T| \\ &= 1 + max(|r'|, |l'|) - (1 + max(|r|, |l|)) \\ &= max(|r'|, |l'|) - max(|r|, |l|) \\ &= \begin{cases} \delta \geq 0, \delta' \geq 0: & \Delta r \\ \delta \leq 0, \delta' \geq 0: & \delta + \Delta r \\ \delta \geq 0, \delta' \leq 0: & \Delta l - \delta \\ \text{否则}: & \Delta l \end{cases}\end{aligned} \tag{B.1}$$

证明. 一次插入不可能同时增加左右分支的高度。平衡因子等于右子树的高度减去左子树的高度。根据其前后变化, 共有四种情况:

1. 如果 $\delta \geq 0$ 并且 $\delta' \geq 0$, 在插入前后, 右子树的高度都不小于左子树的高度。高度的增加全部“贡献”自右子树的变化 Δr;

2. 如果 $\delta \leq 0$, 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增加了, 而左子树保持不变($|l'| = |l|$)。所以高度的增加为:

$$\begin{aligned}\Delta H &= max(|r'|, |l'|) - max(|r|, |l|) && \{\delta \leq 0 \text{ 且 } \delta' \geq 0\} \\ &= |r'| - |l| && \{|l| = |l'|\} \\ &= |r| + \Delta r - |l| \\ &= \delta + \Delta r\end{aligned}$$

3. 如果 $\delta \geq 0$ 且 $\delta' \leq 0$, 和情况二类似, 我们有:

$$\begin{aligned}\Delta H &= max(|r'|, |l'|) - max(|r|, |l|) && \{\delta \geq 0 \text{ 且 } \delta' \leq 0\} \\ &= |l'| - |r| \\ &= |l| + \Delta l - |r| \\ &= \Delta l - \delta\end{aligned}$$

4. 否则 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。高度的增加全部“贡献”自左子树的变化 Δl。

□

B.2 插入后的平衡调整

如图B.1所示, 所有需要修复的四种情况中, 平衡因子都是 ± 2。调整后, 平衡因子 $\delta(y)$ 恢复为 0。左右子树具有相同的高度。

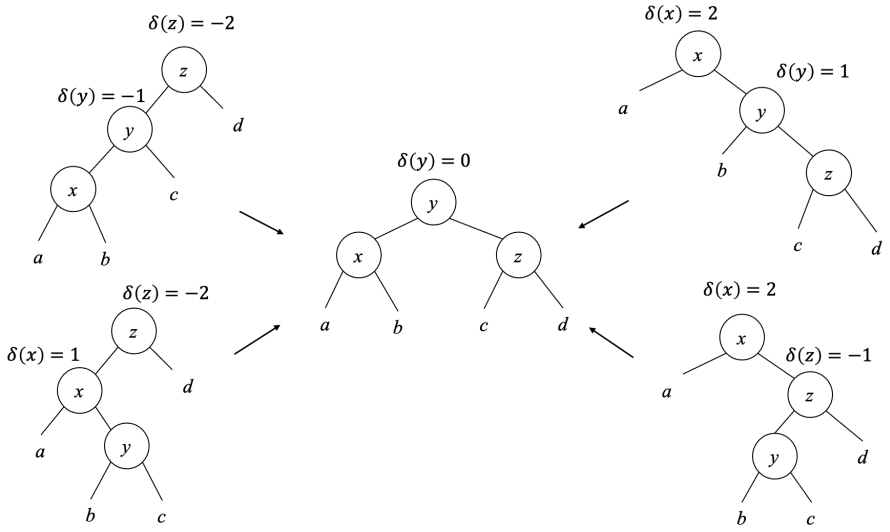


图 B.1: 插入后需要恢复平衡的 4 种情况

这 4 种情况分别是:左−左、右−右、右−左、左−右。记修复前的平衡因子分别为 $\delta(x)$、$\delta(y)$、$\delta(z)$, 修复后分别为 $\delta'(x)$、$\delta'(y)$、$\delta'(z)$。我们接下来证明, 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 $\delta'(x)$ 和 $\delta'(z)$ 的结果。

证明. 我们分别针对四种情况证明。

左−左

子树 x 在调整前后不变, 因此 $\delta'(x) = \delta(x)$。因为 $\delta(y) = -1$ 且 $\delta(z) = -2$, 所以:

$$\begin{aligned} \delta(y) = |c| - |x| = -1 \quad &\Rightarrow \quad |c| = |x| - 1 \\ \delta(z) = |d| - |y| = -2 \quad &\Rightarrow \quad |d| = |y| - 2 \end{aligned} \tag{B.2}$$

调整后:

$$\begin{aligned} \delta'(z) &= |d| - |c| && \{\text{式 (B.2)}\} \\ &= |y| - 2 - (|x| - 1) \\ &= |y| - |x| - 1 && \{x\text{是}y\text{的子节点} \Rightarrow |y| - |x| = 1\} \\ &= 0 \end{aligned} \tag{B.3}$$

对于 $\delta'(y)$, 有如下结果:

$$\begin{aligned}\delta'(y) &= |z| - |x| \\ &= 1 + max(|c|, |d|) - |x| && \{\text{式 (B.3)}, |c| = |d|\} \\ &= 1 + |c| - |x| && \{\text{式 (B.2)}\} \\ &= 1 + |x| - 1 - |x| \\ &= 0\end{aligned} \tag{B.4}$$

汇总上述结果, 对于左－左情况, 新的平衡因子如下:

$$\begin{aligned}\delta'(x) &= \delta(x) \\ \delta'(y) &= 0 \\ \delta'(z) &= 0\end{aligned} \tag{B.5}$$

右－右

右－右和左－左对称, 易知新的平衡因子结果如下:

$$\begin{aligned}\delta'(x) &= 0 \\ \delta'(y) &= 0 \\ \delta'(z) &= \delta(z)\end{aligned} \tag{B.6}$$

右－左

首先考虑 $\delta'(x)$。调整平衡后, 我们有:

$$\delta'(x) = |b| - |a| \tag{B.7}$$

调整平衡前, z 的高度为:

$$\begin{aligned}|z| &= 1 + max(|y|, |d|) && \{\delta(z) = -1 \Rightarrow |y| > |d|\} \\ &= 1 + |y| \\ &= 2 + max(|b|, |c|)\end{aligned} \tag{B.8}$$

因为 $\delta(x) = 2$, 所以:

$$\begin{aligned}\delta(x) = 2 &\Rightarrow |z| - |a| = 2 && \{\text{式 (B.8)}\} \\ &\Rightarrow 2 + max(|b|, |c|) - |a| = 2 \\ &\Rightarrow max(|b|, |c|) - |a| = 0\end{aligned} \tag{B.9}$$

如果 $\delta(y) = |c| - |b| = 1$, 则:

$$max(|b|, |c|) = |c| = |b| + 1 \tag{B.10}$$

将其代入式 (B.9) 得到:

$$\begin{aligned}&|b| + 1 - |a| = 0 \Rightarrow |b| - |a| = -1 \quad \{\text{式 (B.7) }\} \\ &\Rightarrow \delta'(x) = -1\end{aligned} \tag{B.11}$$

反之如果 $\delta(y) \neq 1$, 则 $max(|b|, |c|) = |b|$, 代入式 (B.9) 得到:

$$\begin{aligned} & |b| - |a| = 0 \quad \{\text{式 (B.7)}\} \\ & \Rightarrow \delta'(x) = 0 \end{aligned} \tag{B.12}$$

合并上述两种情况, 我们得到 $\delta'(x)$ 和 $\delta(y)$ 的关系:

$$\delta'(x) = \begin{cases} \delta(y) = 1: & -1 \\ \text{否则}: & 0 \end{cases} \tag{B.13}$$

对于 $\delta'(z)$, 根据定义它等于:

$$\begin{aligned} \delta'(z) &= |d| - |c| && \{\delta(z) = -1 = |d| - |y|\} \\ &= |y| - |c| - 1 && \{|y| = 1 + max(|b|, |c|)\} \\ &= max(|b|, |c|) - |c| \end{aligned} \tag{B.14}$$

如果 $\delta(y) = |c| - |b| = -1$, 则 $max(|b|, |c|) = |b| = |c| + 1$。将其代入式 (B.14) 中得到: $\delta'(z) = 1$。反之如果 $\delta(y) \neq -1$, 则 $max(|b|, |c|) = |c|$, 有 $\delta'(z) = 0$。合并上述两种情况, $\delta'(z)$ 和 $\delta(y)$ 的关系如下:

$$\delta'(z) = \begin{cases} \delta(y) = -1: & 1 \\ \text{否则}: & 0 \end{cases} \tag{B.15}$$

最后, 对于 $\delta'(y)$, 我们可以推导出下面的关系:

$$\begin{aligned} \delta'(y) &= |z| - |x| \\ &= max(|c|, |d|) - max(|a|, |b|) \end{aligned} \tag{B.16}$$

这里又分为三种情况:

1. 若 $\delta(y) = 0$, 说明 $|b| = |c|$, 根据式 (B.13) 和式 (B.15), 有: $\delta'(x) = 0 \Rightarrow |a| = |b|$ 以及 $\delta'(z) = 0 \Rightarrow |c| = |d|$。因此 $\delta'(y) = 0$。

2. 若 $\delta(y) = 1$, 根据式 (B.15), 我们有 $\delta'(z) = 0 \Rightarrow |c| = |d|$。

$$\begin{aligned} \delta'(y) &= max(|c|, |d|) - max(|a|, |b|) && \{|c| = |d|\} \\ &= |c| - max(|a|, |b|) && \{\text{式 (B.13)}: \ \delta'(x) = -1 \Rightarrow |b| - |a| = -1\} \\ &= |c| - (|b| + 1) && \{\delta(y) = 1 \Rightarrow |c| - |b| = 1\} \\ &= 0 \end{aligned}$$

3. 若 $\delta(y) = -1$, 根据式 (B.13), 我们有 $\delta'(x) = 0 \Rightarrow |a| = |b|$。

$$\begin{aligned} \delta'(y) &= max(|c|, |d|) - max(|a|, |b|) && \{|a| = |b|\} \\ &= max(|c|, |d|) - |b| && \{\text{式 (B.15)}: \ |d| - |c| = 1\} \\ &= |c| + 1 - |b| && \{\delta(y) = -1 \Rightarrow |c| - |b| = -1\} \\ &= 0 \end{aligned}$$

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：

$$\begin{aligned}\delta'(x) &= \begin{cases}\delta(y) = 1: & -1\\ \text{否则}: & 0\end{cases}\\ \delta'(y) &= 0\\ \delta'(z) &= \begin{cases}\delta(y) = -1: & 1\\ \text{否则}: & 0\end{cases}\end{aligned} \tag{B.17}$$

左－右

左－右和右－左对称。使用类似的推导，我们可以得到和式 (B.17) 完全相同的结果。

□

B.3 删除算法

删除后会引起子树高度的降低。如果平衡因子超出了 $[-1, 1]$ 的范围，就需要修复以保持 AVL 树的性质。

B.3.1 函数式删除

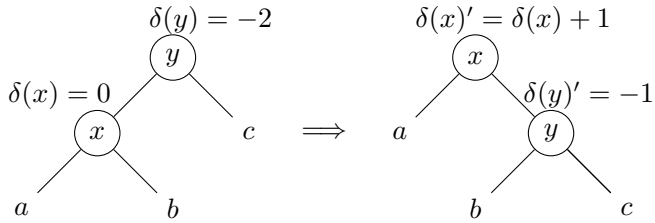
我们先复用二叉搜索树删除算法，然后检查平衡因子并进行修复。删除的结果为一对值 $(T', \Delta H)$，其中 T' 是删除后的新树、ΔH 是高度的减少量。删除算法定义如下：

$$delete = fst \circ del \tag{B.18}$$

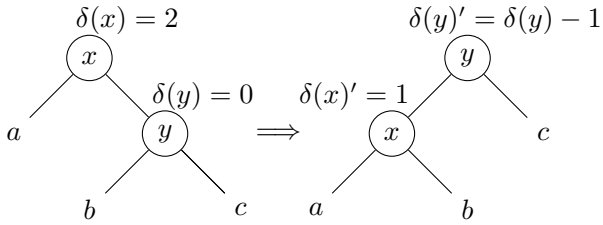
其中 $del(T, k)$ 从树 T 种将元素 k 删除：

$$\begin{aligned} del\ \varnothing\ k &= (\varnothing, 0)\\ del\ (l, k', r, \delta) &= \begin{cases} k < k': & tree\ (del\ l\ k)\ k'\ (r, 0)\ \delta\\ k > k': & tree\ (l, 0)\ k'\ (del\ r\ k)\ \delta\\ k = k': & \begin{cases} l = \varnothing: & (r, -1)\\ r = \varnothing: & (l, -1)\\ \text{否则}: & tree\ (l, 0)\ k''\ (del\ r\ k'')\ \delta\\ & \text{其中 } k'' = min(r)\end{cases}\end{cases}\end{aligned} \tag{B.19}$$

如果树为空，结果为 $(\varnothing, 0)$；否则令树为 $T = (l, k', r, \delta)$。我们比较 k 和 k' 的关系，并沿着子树递归查找和删除。当 $k = k'$ 时，我们定位到了要删除的节点。如果它的任一子树为空，可以将其切下并用另一棵子树替代。否则，将右子树中的最小值 k'' 切



(a) 情况 A



(b) 情况 B

图 B.2: 删除修复

下, 并替换 k'。我们可以复用 $tree$ 函数和 ΔH 的结果。和插入算法相比, 需要增加两种删除中特有的情况:

$$\begin{aligned} & \dots \\ balance\ ((a, x, b, \delta(x)), y, c, -2)\ \Delta H &= (a, x, (b, y, c, -1), \delta(x) + 1, \Delta H) \\ balance\ (a, x, (b, y, c, \delta(y)), 2)\ \Delta H &= ((a, x, b, 1), y, c, \delta(y) - 1, \Delta H) \\ & \dots \end{aligned} \tag{B.20}$$

相应的例子程序如下:

```
delete t x = fst $ del t x where
  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
    | isEmpty l = (r, -1)
    | isEmpty r = (l, -1)
    | otherwise = node (l, 0) k' (del r k') d where k' = min r
```

其中 `min` 和 `isEmpty` 定义为:

```
min (Br Empty x _ _) = x
min (Br l _ _ _) = min l

isEmpty Empty = True
isEmpty _ = False
```

这样总共有 7 种情况需要在 `balance` 中实现:

```
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
-- Delete specific
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

命令式删除使用树旋作来修复平衡。和插入相比, 删除需要处理更多的情况。我们首先复用二叉搜索树删除, 然后再修复子树高度变化引起的平衡问题。

```
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)
```

删除节点 x 时, 记 x 的父节点为 p。如果任一子树为空, 我们将 x 切下, 取代为另一子树。否则, 如果两棵子树都不为空, 我们在右子树中找到最小值节点 z, 将其中的数据复制到 x, 然后将 z 切下。最后, 我们调用 AVL-DELETE-FIX, 并传入根节点 T、

父节点 p、和替换节点 y。记父节点 p 的平衡因子为 $\delta(p)$, 删除后的平衡因子为 $\delta(p)'$。它们之间的关系有三种不同的情况:

1. $|\delta(p)| = 0$、$|\delta(p)'| = 1$。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。修复中止。

2. $|\delta(p)| = 1$、$|\delta(p)'| = 0$。删除前左右子树的高度差为 1, 删除后原来较高的树减小了 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更新树的高度。

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, 我们需要通过树旋转来修复平衡。

对于情况 3, 大部分修复和插入算法相同。我们需要针对图B.2中所示的两种情况进行额外的处理。

```
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
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
24:                     δ_y ← δ( Left(r) )
25:                     if δ_y = 1 then
```

```
                $\delta(p) \leftarrow -1$
            else
                $\delta(p) \leftarrow 0$
            $\delta(\ \textsc{Left}(r)\ ) \leftarrow 0$
            if $\delta_y = -1$ then
                $\delta(r) \leftarrow 1$
            else
                $\delta(r) \leftarrow 0$
        else                                          ▷ 删除特有, 右−右
            $\delta(p) \leftarrow 1$
            $\delta(r) \leftarrow \delta(r) - 1$
            $T \leftarrow \textsc{Left-Rotate}(T, p)$
            break                                     ▷ 高度不再变化
    else if $\delta' = -2$ then
        if $\delta(l) = -1$ then                      ▷ 左−左
            $\delta(p) \leftarrow 0$
            $\delta(l) \leftarrow 0$
            $p \leftarrow l$
            $T \leftarrow \textsc{Right-Rotate}(T, p)$
        else if $\delta(l) = 1$ then                  ▷ 左−右
            $\delta_y \leftarrow \delta(\ \textsc{Right}(l)\ )$
            if $\delta_y = -1$ then
                $\delta(p) \leftarrow 1$
            else
                $\delta(p) \leftarrow 0$
            $\delta(\ \textsc{Right}(l)\ ) \leftarrow 0$
            if $\delta_y = 1$ then
                $\delta(l) \leftarrow -1$
            else
                $\delta(l) \leftarrow 0$
        else                                          ▷ 删除特有, 左−左
            $\delta(p) \leftarrow -1$
            $\delta(l) \leftarrow \delta(l) + 1$
            $T \leftarrow \textsc{Right-Rotate}(T, p)$
            break                                     ▷ 高度不再变化
                                                      ▷ 高度减小, 继续自底向上更新
    $x \leftarrow p$
    $p \leftarrow \textsc{Parent}(x)$
```

```
63:     if p = NIL then                                    ▷ 删除根节点
64:         return x
65:     return T
```

练习 B.1

1. 比较 AVL 树的命令式删除和插入, 将共同的部分抽出, 实现一个通用的 AVL 树修复算法。

B.4 例子程序

下面的例子程序实现了 AVL 树的删除算法:

```
Node del(Node t, Node x) {
    if x == null then return t
    Node y
    var parent = x.parent
    if x.left == null {
        y = x.replaceWith(x.right)
    } else if x.right == null {
        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)
    release(x)
    return t
}
```

其中 `replaceWith` 的定义参见红黑树的部分。`release(x)` 释放节点 x 的空间。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 ≠ null {
        d2 = d1 = parent.delta
        d2 = d2 + if x == parent.left then 1 else -1
        if isLeaf(parent) then d2 = 0
        parent.delta = d2
        p = parent
        l = parent.left
        r = parent.right
        if abs(d1) == 1 and abs(d2) == 0 {
            x = parent
            parent = x.parent
```

```
        } else if abs(d1) == 0 and abs(d2) == 1 {
            return t
        } else if abs(d1) == 1 and abs(d2) == 2 {
            if d2 == 2 {
                if r.delta == 1 {  // 右－右
                    p.delta = 0
                    r.delta = 0
                    parent = r
                    t = leftRotate(t, p)
                } else if r.delta == -1 { // 右－左
                    dy = r.left.delta
                    p.delta = if dy == 1 then -1 else 0
                    r.left.delta = 0
                    r.delta = if dy == -1 then 1 else 0
                    parent = r.left
                    t = rightRotate(t, r)
                    t = leftRotate(t, p)
                } else { // 删除特有，右－右
                    p.delta = 1
                    r.delta = r.delta - 1
                    t = leftRotate(t, p)
                    break // 高度不再继续变化
                }
            } else if d2 == -2 {
                if (l.delta == -1) { // 左－左
                    p.delta = 0
                    l.delta = 0
                    parent = l
                    t = rightRotate(t, p)
                } else if l.delta == 1 { // 左－右
                    dy = l.right.delta
                    l.delta = if dy == 1 then -1 else 0
                    l.right.delta = 0
                    p.delta = if dy == -1 then 1 else 0
                    parent = l.right;
                    t = leftRotate(t, l)
                    t = rightRotate(t, p)
                } else { // 删除特有，左－左
                    p.delta = -1
                    l.delta = l.delta + 1
                    t = rightRotate(t, p)
                    break // 高度不再继续变化
                }
            }
            // 高度减小，继续自底向上更新
            x = parent
            parent = x.parent
        }
    }
    if parent == null then return x // 删除根节点
    return t
}
```

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”. https://en.wikipedia.org/wiki/Function_composition

[9] Wikipedia. “Partial application”. https://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”. https://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”. https://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”. https://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”. https://en.wikipedia.org/wiki/Suffix_tree

[24] Wikipedia. “Trie”. https://en.wikipedia.org/wiki/Trie

[25] Wikipedia. “T9 (predictive text)”. https://en.wikipedia.org/wiki/T9_(predictive_text)

[26] Wikipedia. “Predictive text”. https://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. http://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”. https://en.wikipedia.org/wiki/B-tree

[40] Wikipedia. “Heap (data structure)”. https://en.wikipedia.org/wiki/Heap_(data_structure)

[41] Wikipedia. “Heapsort”. https://en.wikipedia.org/wiki/Heapsort

[42] Rosetta Code. “Sorting algorithms/ Heapsort”. http://rosettacode.org/wiki/Sorting_algorithms/Heapsort

[43] Wikipedia. “Leftist Tree”. https://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”. https://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”. https://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”. https://en.wikipedia.org/wiki/Strict_weak_order

[53] Wikipedia. “FIFA world cup”. https://en.wikipedia.org/wiki/FIFA_World_Cup

[54] Wikipedia. “K-ary tree”. https://en.wikipedia.org/wiki/K-ary_tree

[55] Wikipedia, “Pascal’s triangle”. https://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”. https://en.wikipedia.org/wiki/Tail_call

[62] Wikipedia. “Recursion (computer science)”. https://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 Programming 16:2 (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”. https://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”. https://en.wikipedia.org/wiki/Quicksort

[79] Wikipedia. “Total order”. http://en.wokipedia.org/wiki/Total_order

[80] Wikipedia. “Harmonic series (mathematics)”. https://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”. https://en.wikipedia.org/wiki/Boyer-Moore_string_search_algorithm

[89] Wikipedia. “Eight queens puzzle”. https://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”. https://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”. https://en.wikipedia.org/wiki/Dot_product

[99] Xinyu LIU. “Isomorphism - mathematics of programming”. https://github.com/liuxinyu95/unplugged

索引